9-28-2016

Toward Distributed At-scale Hybrid Network Test with Emulation and Simulation Symbiosis

Rong Rong
Florida International University, rrong001@fiu.edu

DOI: 10.25148/etd.FIDC001176

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

TOWARD DISTRIBUTED AT-SCALE HYBRID NETWORK TEST WITH EMULATION AND SIMULATION SYMBIOSIS

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Rong Rong

2016

To: Interim Dean Ranu Jung
College of Engineering and Computing

This dissertation, written by Rong Rong, and entitled Toward Distributed At-Scale Hybrid Network Test with Emulation and Simulation Symbiosis, 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.

Deng Pan

Raju Rangaswami

Bogdan Carbunar

Gang Quan

Jason Liu, Major Professor

Date of Defense: September 23, 2016

The dissertation of Rong Rong is approved.

Interim Dean Ranu Jung
College of Engineering and Computing

Andrés G. Gil
Vice President for Research and Economic Development
and Dean of the University Graduate School

Florida International University, 2016

ACKNOWLEDGMENTS

I would first like to express my sincere gratitude towards my advisor, Dr. Jason Liu, for his patience and help over the past seven years. Because of his continuous advice, encouragement and necessary support, I am able to go through this doctoral journey and complete my dissertation. I also wish to thank Dr. Deng Pan, Dr. Raju Rangaswami, Dr. Bodgan Carbunar and Dr. Gang Quan, the members of my Ph.D. committee, for their time and meaningful suggestions to support my dissertation.

I further wish to thank my collaborators, Cesar Marcondes and Musa Ahmed, for our illuminating disscussion and their indispensible assistance to my dissertation. I am very fortunate to work in a harmonious and cheerful group. I would like to thank Ting Li, Hao jiang and Mohammad Abu Obaida, for the years we experienced together.

My deepest appreciation is for my parents Delun Rong and Xiaochun Wang. Their endless love and support gave me strength to keep going on this road until today. The last two person I want to thank are my husband Ming Fan and my daughter Iris. Although my little girl is only six months old, she has become the spiritual source of my life. Her little shining face seems to tell me every effort is worthy. As to my husband, I could not express more affection and gratitude for his accompany and encouragement of these years. He take the most part of resposibilities for our family in order to provide me the least burden to finish my dissertation. I owe him everything. Without him, this dissertation would not have been possible.

ABSTRACT OF THE DISSERTATION

TOWARD DISTRIBUTED AT-SCALE HYBRID NETWORK TEST WITH EMULATION AND SIMULATION SYMBIOSIS

by

Rong Rong

Florida International University, 2016

Miami, Florida

Professor Jason Liu, Major Professor

In the past decade or so, significant advances were made in the field of Future Internet Architecture (FIA) design. Undoubtedly, the size of Future Internet will increase tremendously, and so will the complexity of its user behaviors. This advancement means most of future Internet applications and services can only achieve and demonstrate full potential on a large-scale basis. The development of network testbeds that can validate key design decisions and expose operational issues at scale is essential to FIA research. In conjunction with the development and advancement of FIA, cyber-infrastructure testbeds have also achieved remarkable progress. For meaningful network studies, it is indispensable to utilize cyber-infrastructure testbeds appropriately in order to obtain accurate experiment results. That said, existing current network experimentation is intrinsically deficient. The existing testbeds do not offer scalability, flexibility, and realism at the same time. This dissertation aims to construct a hybrid system of conducting at-scale network studies and experiments by exploiting the distributed computing ability of current testbeds.

First, this work presents a synchronization of parallel discrete event simulation that offers the simulation with transparent scalability and performance on various high-end computing platforms. The parallel simulator that we implement is auto-configured so that it can self-adapt the performance while running on supercomputers with disparate architectures. The simulator could be used to handle models of different sizes, varying modeling details, and different complexity levels.

Second, this work addresses the issue of researching network design and implementation realistically at scale, through the use of distributed cyber-infrastructure testbeds. An existing symbiotic approach is applied to integrate emulation with simulation so that they can overcome the limitations of physical setup. The symbiotic method is used to improve the capabilities of a specific emulator, Mininet. In this case, Mininet can be used to run applications directly on virtual machines and software switches, with network connectivity represented by detailed simulation at scale. We also propose a method for using the symbiotic approach to coordinate separate Mininet instances, each representing a different set of the overlapping network flows. This approach provides a significant improvement to the scalability of the network experiments.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

FIGURE PAGE

CHAPTER 1

INTRODUCTION

In the past decade, significant advances have been made in Future Internet Architecture (FIA) design. Undoubtedly, the size of Future Internet will increase considerably, as will the complexity of its user behaviors. This growth implicates most of future Internet applications and services can only achieve their full potential on a large-scale basis. Researchers will not be able to extrapolate macroscopic network effect and behavior simply from an analytical model, or from experiment results of small-scale networks. Thus the development of network testbeds that can validate key design decisions and expose operational issues at scale is essential to the FIA research. Along with the development of FIA, cyber-infrastructure testbeds have also achieved remarkable progress. For example, the Global Environment for Network Innovations (GENI) has been a community-based effort for building a collaborative and exploratory network experimentation platform [GENa]. While network researchers and engineers can validate design and implementation directly on the cyber-infrastructure testbeds, the inherent deficiencies of solely relying on real-world implementation and physical deployment in network studies must be avoided. For meaningful network studies, it is indispensable to obtain accurate experiment results under various network conditions, which can be achieved through the appropriate use cyber-infrastructure testbeds appropriately.

This dissertation aims to construct a hybrid system of conducting at-scale network studies and experiments by exploiting the distributed computing ability of current testbeds. This system effectively combines distributed emulation and simulation, through applying a symbiotic method [EL13], in order to simultaneously provide two significant properties. The first property is scalability and flexibility. Although federated platforms offer researchers additional computing and network resources to perform experiments through resource slicing, it must be noted that there still exists a physical limit either the network size or the traffic volume. Another shortcoming of federated platforms is their inflexibility when it comes to changing the existing setup. As previously mentioned, the most of the future applications can only unveil specific properties under large-scale experiments. In order to reveal scaling properties and robust issues, parallel simulation must represent a necessary and significant part of the system. This work starts by examining the synchronization problem in parallel simulation, which is a traditional topic in this area of research. In most cases, network researchers are unaware of the type of resources allocated for their tests, especially since the availability of these resources and their allocation may vary with time. Additionally, the nature of these resources (virtual or physical) is unknown to the researchers. For these concerns, a self-adaptive synchronization algorithm is designed with the aim of achieving optimal performance for various models on heterogeneous computing platforms. This algorithm is implemented in a minimalistic parallel simulator that can efficiently run on supercomputers up to thousands of cores with power speedup. The second property is the trade-off between realism and scalability, which is is a constant theme in networking experimentation. This work applies a symbiotic approach to organically combine network emulation and network simulation. In particular, Mininet [LHM10, HHJ^{+}12] is the emulator that was chosen to be improved due to its potential to validate the design and operational issues of OpenFlow applications. In this case, Mininet can be used to run real applications directly on virtual machines and software switches while embedding in diverse simulated network settings. In addition to the ability of representing complex networks by parallel simulation at scale, this work also proposes a method to coordinate several Mininet instances running on distributed machines. Through the representation of a different set of the overlapping real flows on each instance, the traffic volume can be significantly increased. Network experiments, through this approach, can and do achieve a good balance among flexibility, scalability, and accuracy.

1.1 Motivation

As the size of Internet increases, efficiently studying the behavior of the network becomes increasingly challenging. Some applications only show full potential on large-scale, for example, scalability is critical for peer-to-peer since it exhibits ”network effect”, which means that the behavior of one user is positively affected when another user joins and leaves the network [RFI02]. Another example is worm study, which requires a large-scale model to show the propagation dynamics of the worm [LYPN02]. Due to the network complexity, there are no analytical models that can accurately describe the behavior of network applications under a wide variety of large-scale conditions. This fact makes large-scale network simulation an indispensable tool for studying immense networks. In certain cases, only large-scale simulation is able to gain credible evidence. However, it is challenging to represent the behavior of network applications under a wide variety of large-scale traffic conditions in a repeatable and controllable fashion. By conducting a survey of SIGCOMM papers from the year 2007 to 2013 [LL14] we were able to identify the following: because of the lack of large-scale tools, researchers often use results from small-scale network simulation to extrapolate large-scale network simulation. Therefore, for exploring future network design, a simulator capable of accurately reproducing large-scale and dynamic network behaviors would be highly valuable.

Parallel discrete event simulation (PDES) has been broadly used in network studies and has demonstrated its capability of simulating large-scale models. However, the synchronization between different computing units has been the critical hurdle for the performance of parallel simulation. There are efforts of federated testbeds for network experiments in the recent development of cyber infrastructure. These testbeds like GENI [GENa], CloudLab [Clo], and Chamelon [Cha] manage and interoperate different types of resources provided by various organizations, so as to support multiple independent users in their experiments by resources slicing. Because of the unknown types of resources available for users, particularly virtualized ones, the synchronization design is more complicated than before. The quality of parallel synchronization is measured by its ability to make the simulator run transparently with good performance on various testbeds.

Existing network testbeds offer different capabilities, in terms of *realism*, *scalability* and *flexibility*. Simulation is the best choice for experiments that emphasize the scale, complexity and dynamics of network experiments. Network simulation [NS-a] can be effective at capturing overall design aspects, answering what-if questions, and revealing complex system characteristics. The scale of network models can increase several magnitudes by using parallel simulation. However, the start point for network experiments is to validate design and implementation issues; the fidelity cannot be totally ignored in this context. In this aspect, simulation exposes the deficiency for lack of a certain level of realism; in contrast, physical testbeds fit in. Live experiments running directly on physical testbeds [PACR02, Pla] provide realistic environments and traffic conditions. In physical experiments, scalability and flexibility will be sacrificed, because it is hard to overcome the physical limits or change the hardware configurations. Thus networking researchers often use simulation as a complementary role for experimenting on physical testbeds. In most of the cases, they are used in an isolated fashion.

The essential constraint for the scalability of using physical testbeds is—it is unrealistic to handle all-to-all traffic in live networking experiments. In networking studies, the bulk of traffic generated by other applications, usually called *background traffic*, has a significant influence on the traffic generated from the target application. Since the generation of this portion of the traffic in live experiments is unrealistic and unattainable, correctly reflecting its effect is essential for truthfully evaluating the performance of a new application or protocol. In this case, network emulation is the best available choice. Emulation testbeds provide the native operating system of target applications while mimicking network between application instances. The states of network links, such as delays and drop probabilities, can be regulated in the emulation. This feature provides more operational capabilities. Although emulation provides the flexibility of embedding real applications in various test scenarios, it is still limited in scale and in the traffic handling capacity. An ideal network experiment would test real target applications directly in a large background network with diverse scenarios and traffic conditions. It is a nontrivial task for the testbeds to run large-scale experiments without losing realism.

As mentioned earlier, the development of cyber-testbeds paves the way for network researchers to validate design and implementation details. To fully utilize the computing power, the experimental system must be capable of running the tests on any and all types of machines. Another necessary feature for the system to have is the potential of evaluating future network design and architecture.

1.2 Problem Definition

To construct a hybrid system of conducting at-scale network studies and experiments by exploiting the distributed computing ability of current testbeds, we formalize two problems we intend to address in this dissertation.

- **Efficient parallel simulation for large-scale networks.** In order to perform large-scale experiments and offer diverse network scenarios, we need to utilize parallel simulation. However, the scale and complexity of nowadays systems grow rapidly, as does the variety of the models, large or small, complicated or simple, in distinctive areas. Meanwhile, the development of diversified high-end computing platforms also moves at a similar fast pace. Therefore, the objective is to conduct efficient parallel simulation for different models and ensure that they run transparently on multiple supercomputers by harnessing their parallel capabilities to cope with large-scale models.

- **Realistic at-scale network experiments with federated testbeds.** As previously mentioned, experiments must be carried out realistically in order to fulfill the purpose of validation. The realistic property can be achieved by harnessing the capability of distributed physical platforms. For accommodating flexibility, scalability and compatibility, network researchers have to overcome the hardware limitations of physical testbeds. Representing intensive *background traffic* in real experiments is also challenging. Therefore, the goal is to build a hybrid system that can analyze target applications *in-situ* with needed operational realism and with live network traffic conditions, by embedding them seamlessly in dynamic, large-scale networks with cross-traffic from other applications. The execution compatibility on heterogeneous platforms, and the future extendibility will also be the necessary feature for our system.

  The overarching goal of our work is to provide a hybrid system so that we can perform large-scale network experiments and studies under diverse scenarios. This system must execute on any distributed platforms.

1.3 System Design and Contributions

In this section, we describe the overall design of our hybrid experimentation system. We also provide a description of the system's subcomponents, each tackling a stated problem, to achieve the ultimate goal. Below, the major contributions of this work are listed.

Fig. 1.1 represents the design of our hybrid system. The system is composed of distributed simulation and distributed emulation, a combination that simultaneously offers *realism*, *scalability* and *flexibility* for network experiments.

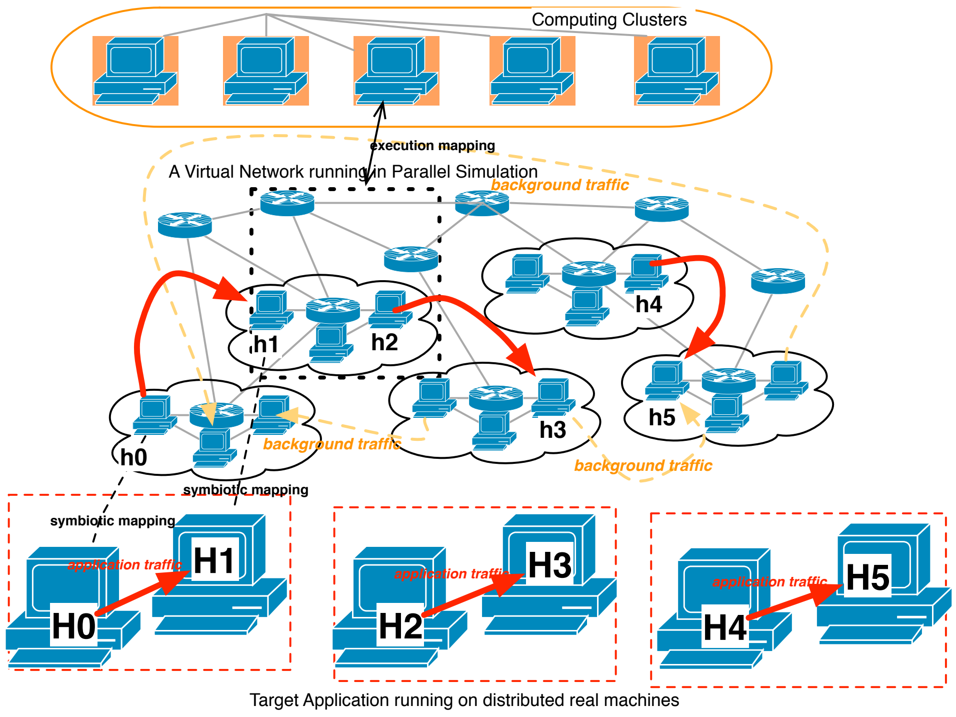


Figure 1.1: Overall system design.

1. First of all, we need a self-adaptive synchronization so as to make parallel simulation scale up ideally, regardless of the types of platforms.

    - We propose a synchronization algorithm for parallel discrete-event simulation, called hierarchical composite synchronization. The approach is extended from composite synchronization [NL02] to avoid performance pitfalls of two traditional synchronization methods in parallel simulation. In particular, our hierarchical method addresses the discrepancy in the communication and synchronization cost for shared-memory multiprocessor multicore machines and distributed-memory machines.

    - We implement the algorithm in a parallel simulator, called *MiniSSF*. It is a simplified and yet more streamlined implementation of a Scalable Simulation Framework (SSF) [CNO99], which had been widely adopted for parallel simulation in many areas. *MiniSSF* removes some redundant complex components, and also introduces several new features for improving the operational capability and performance of the simulator. We conduct extensive experiments that can demonstrate the simulator ability to achieve scalability by running on thousand of cores. From the experiments, we believe that *MiniSSF* is able to run large-scale models and self-adjust the synchronization configuration for achieving better performance on various platforms.

2. Second, we need to organically integrate emulation with simulation, and map it onto distributed cyber-infrastructure testbeds. [1]

   - We apply a symbiotic approach [EL13] to combine simulation and emulation on a specific network emulator, Mininet. This approach makes simulation and emulation form a symbiotic relationship allowing them to benefit from each other. Mininet is a container-based emulator combined with software-switches that can emulate small networks on a laptop. With the hybrid system *mininet-symbiosis*, one can use Mininet to directly run applications on virtual machines and software switches, with network connectivity represented by detailed simulation at scale.
   - We also present a method for using the symbiotic approach to coordinate separate Mininet instances to run (with different virtual machines and switches) on distributed machines. Thus the system can be mapped to any cyber-infrastructure platforms. In this case, the scalability of the network experiment can be significantly improved. One can conduct hybrid at-scale tests for validating different design parameters easily on various platforms by using this approach.

[1]The system should also be considered to execute on virtualized computing resources.

This dissertation work is drawn from the following papers:

- Hierarchical Composite Synchronization, J. Liu and R. Rong, PADS 2012
- Performance Study of Minimalistic Simulator on XSEDE Massively Parallel Systems, R. Rong, J. Hao and J. Liu, XSEDE 2014
- Symbiotic Network Simulation and Emulation, M. Erazo, R. Rong and J. Liu, Transactions on Modeling and Computer Simulation, TOMACS 2015
- Toward Scalable Emulation of Future Internet Applications with Simulation Symbiosis, J.Liu, C. Marcondes, M. Ahmed and R. Rong, DS-RT 2015

1.4 Outline

The rest of this dissertation is organized as follows. In Chapter 2, we introduce the existing networking experimental testbeds. We review different experimenting methods and discuss the trade-off between flexibility, scalability, and fidelity of the current testbeds. We also review the related work of parallel discrete event simulation (PDES), and describe synchronization methods in PDES especially. Additionaly, we describe the existing symbiotic network studies. At last, we present the development of recent cyber-infrastructure testbeds.

In Chapter 3 we propose a hierarchical composite synchronization algorithm for parallel discrete-event simulation. The algorithm is designed to address the discrepancy in the communication and synchronization cost for shared-memory multiprocessor multi-core machines and distributed-memory machines. This approach allows the simulator to fully exploit parallelism for various models, and to self-adjust the performance for diverse computing platforms. We also present our work of implementing the new synchronization in a minimalistic parallel simulator in this chapter. In particular, the simulator incorporates several new approaches in order to improve the scalability of models, the ease to use and the compatibility with different platforms. We conduct extensive experiments to show that our simulator can achieve scalability and good performance on a variety of heterogeneous parallel computing platforms.

In Chapter 4 we present our hybrid system of applying a symbiotic approach to combine network simulation and emulation on a specific emulator, Mininet. We start by reviewing the idea of the symbiotic method. We highlight how it can overcome the intrinsic deficiencies of existing experimentation approaches. We then describe, in detail, how to our system *mininet-symbiosis* is implemented from several important aspects. Finally, we propose a method for using the symbiotic approach to coordinate separate Mininet instances, each representing a set of different yet possibly overlapping network flows. We provide a prototype implementation of the distributed hybrid system and present validation studies to show it can achieve accurate results. We also present a case study that successfully replicates the behavior of a denial-of-service (DoS) attack protocol.

Finally, in Chapter 5 we present the conclusions of our research and discuss future research directions.

CHAPTER 2

BACKGROUND

In this chapter, we describe the most relevant literature to provide the state of art. We first review existing experiment network testbeds in section 2.1. In particular, we review several parallel simulators, as well as the widely used synchronization techniques in parallel simulators in section 2.2. Second, we review examples of symbiotic systems used for network experimentation 2.3. Finally, we briefly review recent development of cyber-infrastructure in 2.4.

2.1 Network Testbeds

There are three basic types of network experiment testbeds: physical, simulation, and emulation. The choice of using network testbeds largely depends on the goal of the study. On the one hand, physical and emulation testbeds can execute real applications, operate with real systems, accept real input, produce real output, and respond to real network conditions. They provide the operational realism and fidelity usually unattainable by modeling and simulation. On the other hand, simulation is expedient for constructing and testing models to obtain ”the big picture”, which should be highly valuable especially when a good understanding of the system’s complex behavior is absent. Simulation makes it easy for prototyping, for exploring the design space, for assessing the performance in diverse network settings, and for investigating what-if scenarios.

Physical testbeds provide a real networking environment for live experimentation. Physical testbeds can be further divided into production testbeds and reconfigurable testbeds. Production testbeds (such as Internet2 [Int] and ESnet [ESn]) support live network experiments; however, they allow only “safe” experiments that do not disrupt normal operations, and they provide only a small and iconic version of the entire internet. Comparatively, reconfigurable testbeds provide far better flexibility. PlanetLab [PACR02] is a well-known reconfigurable testbed consisted of machines distributed across the internet and shared by researchers simultaneously conducting multiple experiments. An experiment can run on a subset of machines creating an overlay network (called a slice). Although reconfigurable, it is difficult to test applications beyond the existing setup and configuration of the underlying physical environment, which is limited in scale and capacity. It would be difficult to realize experiments with the number of nodes significantly larger than the available nodes (either physical or virtual machines), and with the capacity of interconnectivity higher than the available bandwidth.

Emulation testbeds support ”traffic shaping” by introducing artificial packet delays and packet losses [Riz97]. They can be built on a variety of computing infrastructures, 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]). Mininet [LHM10] is a recent emulation testbed using Linux containers and traffic control (`tc`). One should note that both physical and emulation testbeds can only support experiments of limited scale, due to resource limitation and heavy resource sharing. For example, we observe that there exists a stringent limitation in the amount of traffic that can be emulated in real time. The aggregate traffic on each physical machine cannot go beyond a certain rate, which depends on the machine type (typically, a few gigabits per second).

Mininet [LHM10, HHJ$^+$12] is a popular container-based emulation environment built on Linux for testing OpenFlow [MAB$^+$08] applications. Using Mininet, one can create network experiments using a set of virtual hosts and virtual switches connected as an arbitrary network. Mininet uses the native Linux namespaces to represent virtual hosts. It is a lightweight container-based virtualization solution, based on which one can create relatively large virtual networks with hundreds and even thousands of virtual machines on a single physical machine. The containers can be connected to the instances of the Open vSwitch (OVS) [ovs], which is a production-quality software switch augmented with OpenFlow capabilities for experimentation with SDN applications. However, the traffic between the physical machines has to be limited by the available connection bandwidth. For experiments that induce heavy traffic, Mininet cannot produce reliable results. As the virtualization technique becomes mature in these years, several dedicated emulation systems have been tried to scale up by multiplexing virtual nodes on a single physical machine, and multiplexing virtual links on a physical network link [HRS+08, ADHK08]. Others original container-based emulation have been extended to run on a cluster [WDS+14, RBZ+14]. However, all these efforts are designed as a top-down approach, which starts from partitioning the network on to physical or virtual computing resources. Therefore, resource allocation for resolving contention plays a decisive role for their performances. One should also realize the configuration and the administration of these systems may introduce issues. Maxinet [WDS+14] is a success for the distributed extension of Mininet, whereas has picky requirements for underlying platforms.

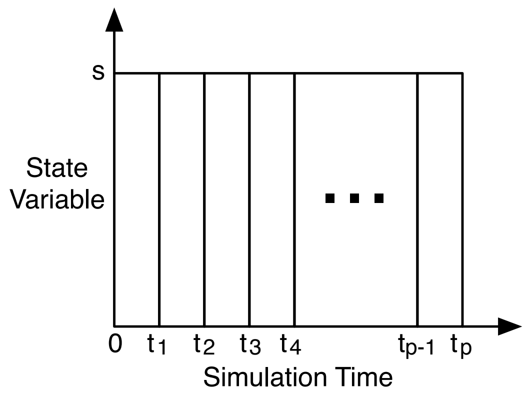
Simulation testbeds (NS2 [NS-a], NS3 [NS-c], OPNET [OPN], OMNeT++ [VH08]) provide better flexibility and controllability as they contain only software modules for characterizing network operations. Simulation may lack realism, and therefore would typically require extensive efforts in validation. Developing detailed models is also known to be labor-intensive. For dealing with these issues, one way is to directly incorporate protocol implementations in simulation [LXC04, LYN+05, TUM+13]. This technique is called *direct-execution simulation*, which includes compile-time techniques (which involve little or only moderate modification to the source code), link-time techniques (such as using linker wrapper functions to replace functions related to communication and timing), and run-time techniques (such as binary code modification, preloading dynamic libraries, or using packet capturing facilities). There are two major issues with this approach. First, reproducing detailed behavior for all network protocols and applications in the simulation would be too costly to realize for full-scale network experiments. Second, in cases where one may desire high-level models, such as random traffic generation and stochastic failures, implementing detailed network models does not automatically translate to an accurate representation of high-level behaviors.

2.2 Parallel Simulation

Parallel simulation is a technique of running a single discrete-event simulation program in parallel [Fuj90]. It can harness the collective power of parallel computers to run complex large-scale models and thus can be successfully applied to increasing the performance and scalability of network simulations, e.g., SSFNet [NLLY03], GTNets [Ril03], ROSSNet [CBP00a], and GloMoSim [BTA+99]. Simulation can be effective at capturing large-scale system design, and answering what-if questions. With parallel simulation, one is able to handle very large-scale models.

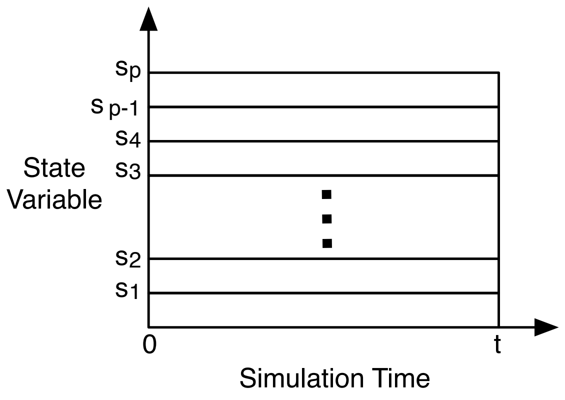
2.2.1 Parallel Discrete Event Simulation (PDES)

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, which can significantly improve the performance of simulation. 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 $[t_{p-1}, t_p)$, 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



b

(a) Time-parallel decomposition.

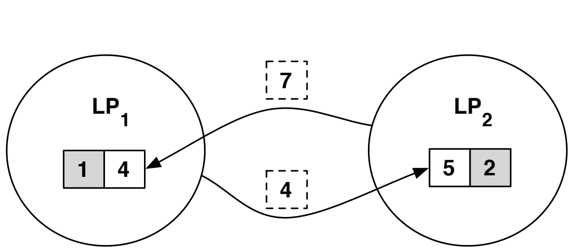


(b) Space-parallel decomposition.

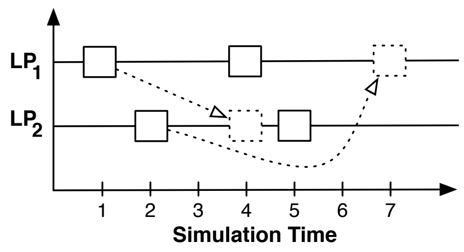
Figure 2.1: Decomposition of simulation models for parallel execution.

the states at the time boundaries ($\{t_1, t_2, \ldots, t_{p-1}, t_p\}$), 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 simulate 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 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



(a) Logical Processes



(b) Timeline

Figure 2.2: Example process-oriented simulation.

final result is the same as if the model was 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 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 situation is the heart of the synchronization problem for PDES frameworks.

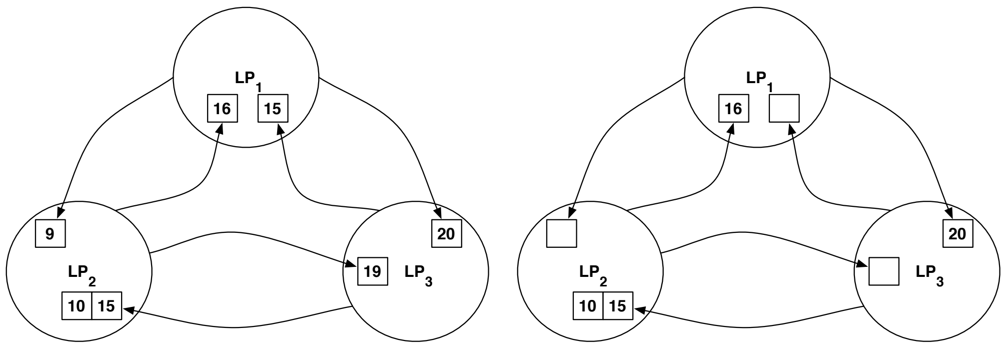
2.2.2 Synchronization in Parallel Simulation

Discrete-event simulation needs to execute events in a non-decreasing timestamp order to ensure causality. The fundamental issue of parallel simulation is therefore how to synchronize the LPs so as to preserve the timestamp order execution of events at each LP. Two classes of protocols have been developed for synchronization in parallel simulation. The optimistic approach permits out-of-order event execution: when the simulation detects a causality error, it will rewind the simulation and roll back the erroneous computations [Jef85]. The conservative approach prohibits out-of-order event execution: an LP must be blocked from executing its next event unless it is guaranteed not to induce causality errors. Details of both approaches are below.

Conservative Synchronization

In conservative method, synchronization needs to take place among the LPs to ensure that no event with a timestamp smaller than that of the next event will arrive at an LP in the future.

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 has 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



(a) Logical Processes (b) Deadlocked Logical Processes

Figure 2.3: Example of how logical processes get deadlocked.

lowest timestamp is to wait for an event to arrive at all of its queues. It 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. The particular arrangement produces the outcome in figure 2.3(b) where each logical process is waiting for another logical process before it can proceed, resulting in a deadlock.

The CMB algorithm avoids deadlock using null messages. A *null message* is sent over a channel in lieu 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. Before execution, each LP needs to determine the lower bound on the timestamp (LBTS) of future events to arrive at the LP by scanning through its incoming channels. LBTS is the upper bound in simulation time up to which the LP is able to safely advance its clock. Once the clock has indeed been updated, the LP sends a null message through each of the outgoing channels, which can potentially increase LBTS of the successor LPs and unblock them. 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 significantly decreased. There have been many extensions to the classic CMB algorithm to improve its efficiency. For example, on shared-memory multiprocessors, the null-message protocol can be replaced with an LP scheduling algorithm, such as the Critical Channel Traversing (CCT) algorithm [XUSC99], which selects the ready LPs to run on parallel multiprocessors (each with LBTS larger than it local simulation clock). All such algorithms prevent the deadlock in an asynchronous manner, so they are classified in asynchronous synchronization. However, for all algorithms in this category, each time an LP is scheduled for execution, it makes at least one scan through its incoming channels and one scan through its outgoing channels; thus, the cost is proportional to the node's degree in the LP graph.

Synchronous algorithms form another class of conservative synchronization protocols by making use of collective operations, such as barriers and min-reductions. For example, the YAWNS protocol [Nic93] uses global barriers to delineate the synchronization windows, within which the LPs are safe to process events without introducing causality errors. The size of the synchronization windows are determined by the worst-case lookahead among all LPs. Chandy and Sherman's Conditional Event approach [CS89] also uses a global min-reduction to determine the lower bound on the timestamp of all LPs to safely process events, calculated from the timestamp of each LP's earliest conditional event plus the lookahead. Likewise, other synchronous approaches, including Lubachevsky's Bounded Lag algorithm [Lub88] and Ayani's Distance Between Objects [Aya89], all depend on collective operations to help find LBTS for all LPs. Although synchronous algorithms are simple and scalable (given the logarithmic cost of the barrier and min-reduction operations), the cost is proportional to the frequency of performing the global operations.

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 affected 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.

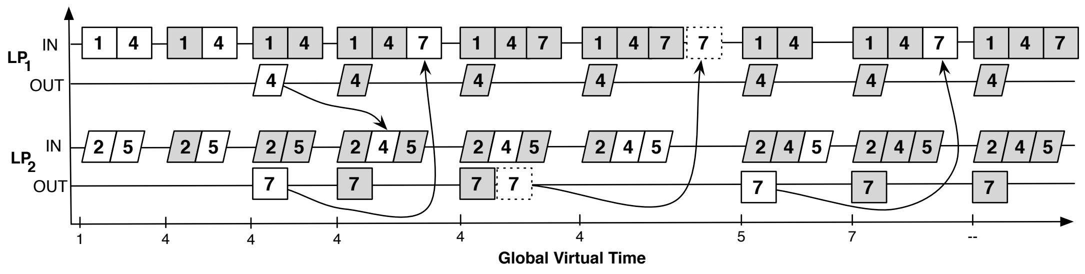


Figure 2.4: Event cancellation example.
Example of how TimeWarp cancels events in response to encountering a straggler event.

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.

Although the optimistic approach can be made fully automatic, it has complications related to performance and model complexity. For example, state saving introduces memory overhead, staggering rollbacks can cause cascading effects, and out-of-order event execution may introduce unexpected faulty conditions [Fuj90, NL97]. There is a large and rich body of work improving and augmenting TimeWarp and other algorithms in a similar vein.

2.2.3 Parallel Simulators

There have been numerous parallel simulators built to conduct performance studies. Some are of general purpose; others are domain-specific. For example, number of parallel simulators are designed to model computer networks. Here we focus on a few simulators capable of running large-scale models that have gained widespread used= by the research community.

PDNS

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. NS-2 is prevelant because that is easy to use and it has a rich collection of protocols. However, the magnitude of models can be running on NS-2 can only reach a thousand nodes [NS-b]. It makes NS-2 can not exploit the capability of nowadays powerful parallel computing platforms.

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 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].

SSFNet

SSFNet [NLLY03, Ren] 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 the 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 computer to be outside the simulation environment. This diverges from other contemporary PDES frameworks such as GTNets [Ril03] or PDNS [FPP+03].

2.3 Symbiotic Testbeds for Networking Experimentation

2.3.1 Real-time Simulation and On-line Simulation

There are two promising areas that combine network simulation and emulation. *On-line simulation* uses simulation as an integrated service for real-time network management with the goal of improving network performance, via network planning, monitoring, parameter tuning, and traffic engineering (e.g., [SSS+02, YKH+01]). *Real-time simulation* performs simulation in real time so that the target virtual network can interact with real network entities (e.g., [Fal99, BSU00, ZJTB04, LLN+05, ADHK08, LLV+09, NJZ11]).

Real-time simulation aims to create an accurate, scalable and flexible networking testbed by combining large-scale network simultaion with emulation. To support real-time simulation, the simulator is modified to be able to regulate the virtual time advancement; in parallel simulation, the issue becomes an effective scheduling of the logical processes with respect to real time [Liu13]. Although real-time simulation allows hybrid network experiments involving both simulated and physical network components, the scale of the network experiments is constrained by the I/O capacity of the simulator for exchanging network packets with the physical system [LLV$^+$09]. Real-time simulation could be treated as a significant foundation of our work. We review several real-time simulators in the reminder of this section.

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

The Real-time Immersive Network Simulation Environment (RINSE) [LLN^{+}05] extends DaSSF [LN] 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.

However, traffic exchanging between simulation and emulation is in real packets for most real-time simulators. This non-negilgible synchronization overhead becomes performance bottleneck for real-time simulation.

2.3.2 Existing Symbiotic Approaches

In Biology, symbiosis is defined as the mutually beneficial relationship between two or more different organisms. Symbiotic simulation can be defined as "one that interacts with the physical system in a mutually beneficial way" [FLPU02]. From the definition of symbiosis, it is clear symbiotic simulation systems are different from general discrete-event simulations. In general simulations, all parameters need to be configured before simulation, and no changes can be made during runtime. Two types of symbiotic systems are defined in [ATCL08], both of which create feedback control loop between simulation and physical system. *Closed-loop symbiotic simulation systems* perform what-if experiments based on scenarios retrieving from physical system and provide decisions to control the behavior of physical systems.*Open-loop symbiotic simulation systems* only draw information from physical systems but do not provide feedbacks. These systems are used to describe current states, predict future behaviors, and detect anomalies of physical systems.

Symbiotic simulation is also referred as DDDAS (Dynamic Data-Driven Application System), in a larger context that has broadly applied in the areas of manufacturing, business, system engineering, civil engineering, biology, social science, and many other disciplines. In DDDAS, simulation and the physical system form a symbiotic feedback control system, whereas a simulation can dynamically incorporate data from the physical system so that it can improve the measurement process or exercise more precise control of the physical system [DDD14].

ROSENET

ROSENET [GF09, Gu07] is an early attempt to promote the symbiotic relationship between simulation and emulation. It combines a high-performance simulator and a low-fidelity emulator running at separate locations. The simulator continuously updates the emulator with link statistics, including packet delay, jitter, and loss. The emulator also continuously updates the simulator with a summary of the real traffic.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.

ROSENET is based on the findings that the Internet traffic exhibits a constancy in timescales of minutes [ZD01]. ROSENET aimed to achieve *accuracy* through the use of emulators. Allowing users to run arbitrary network topologies and traffic loads is to provide *flexibility* and *scalability*. It also provides *accessibility*, since real world applications can run locally while the simulator can run on remote high performance facilities. Although ROSENET achieved initial success, it is shown to be capable of emulating only a single bottleneck link and also only applications that generate non-responsive traffic (i.e., UDP applications).

2.4 Recent Development of Cyber-infrastructure

The Global Environment for Network Innovations (GENI) has been a community-based effort for building a collaborative and exploratory network experimentation platform for studying future network applications [GENa]. Follow-up efforts include various cyber-infrastructure design, development, and build-out projects, such as NSFCloud [Clo, Cha], for building mid-scale cloud-computing testbeds in the U.S. There are similar attempts made in European Union, Japan, Brazil, and other nations. We focus on introducing GENI and NSFCloud in following.

GENI [GENa] is a set of network research infrastructure, which aims to be presented as a single collaborative and exploratory platform for implementing and testing new network designs and technologies. GENI offers several features: i) *deep programmability*: researchers can program not only the end hosts of your experimental network but also the switches in the core of your network ii) *a large-scale experiment infrastructure*: GENI gives researchers access to hundreds of widely distributed resources including compute resources such as virtual machines and bare-machines, and network resources such as links, switches and WiMax base stations. iii) *controllability*: Everyone can get exclusive access to certain GENI resources including CPU resources and network resources. Each user shall be provided with a *slice*, i.e., a subset of resources of the GENI infrastructure, and network experiments shall be conducted independently on reserved resources within slices. The current GENI design consists of three main types of entities: *clearinghouses*, *aggregates*, and *principals*. A clearinghouse is a central location for management of GENI resources for experimenters and administrators. Specifically, it provides registry services for principals, slices and aggregates, and authentication services for accessing the resources. An aggregate represents a group of components encapsulating the GENI sharable resources (including computation, communication, measurement, and storage). When an experimenter from a research organization (i.e., a principal) decides to conduct a GENI experiment, she will negotiate with the clearinghouse and the associated aggregate managers through an elaborate resource discovery and allocation process. In response to the experimenter's request, each participating aggregate will provide a set of requested resources, which are called *slivers*. Jointly, these slivers form a *slice*, which is the environment where the experimenter conducts experiments, with the help of GENI experiment support services.

Chamelon [Cha] is the first of NSFCloud projects, which aims to provide a large-scale, reconfigurable experimental environment for cloud research. Researchers are able to configure slices of Chamelon as custom clouds using pre-defined or custom software to test the efficiency and usability of different cloud architectures on a range of problems, from machine learning and adaptive operating systems to climate simulations and flood prediction. Chamelon benefits users to run experiments on a large-scale, critical for big data and big compute research. Another aspect of Chamelon is its support for heterogeneous computer architectures, which provide the capabilities to researchers to mix-and-match hardware, software and networking components to conduct performance studies. The second NSFCloud project CloudLab [Clo], is a large-scale distributed infrastructure supporting OpenFlow and other software-defined networking technologies.

CHAPTER 3

A SELF-ADAPTIVE PARALLEL SYNCHRONIZATION

In this chapter, we present a synchronization algorithm for parallel discrete-event simulation, called hierarchical composite synchronization. Our design idea is to make parallel synchronization self-adaptive its performance to different models and platforms. The new algorithm is extended from composite synchronization [NL02] that combines an asynchronous CMB-style channel scanning method with a synchronous window-based method to avoid pathological situations where neither synchronization algorithms would perform optimally. We also provide an analytical model to predict the performance of hierarchical composite algorithm. Through experiments, a significant performance improvement has been observed of the new algorithm over different combinations of the traditional asynchronous and synchronous approaches used separately for distributed-memory and shared-memory of distinctive supercomputers.

In the section 3.6 of this chapter, we also present the details of a parallel simulator that implements the new synchronization algorithm. It is a simplified and yet more streamlined parallel simulator, called *MiniSSF* [RHL14]. It starts from a widely adopted parallel simulation API, Scalable Simulation Framework (SSF) for large-scale discrete-event models. *MiniSSF* maintains salient features from SSF, makes improvements for deficiencies, and incorporate new ideas and technics to achieve better performance.

3.1 Introduction

Synchronization is a fundamental issue of parallel discrete-event simulation (PDES). An important aspect of synchronization is the simulator's ability to effectively exploit parallelism in large complex models without overly tasking the modelers who are expected to be knowledgeable in their respective domains and less concerned with parallel computing problems. We have already reviewed two primary parallel synchronization categories in chapter 2. In this chapter, we simply describe the property of existing synchronization algorithms, and majorly introduce how does our algorithm make improvements over the existing ones.

Optimistic synchronization, which allows logical processes (LPs) to advance simulation without the presumptuous articulation of causality, has been considered as the most promising solution. The problem, however, is that the ability to rewind the simulation clocks and roll back erroneous computations requires sophisticated mechanisms, such as state saving [Jef85] and reverse computation [CPF99]—both turn out to be heavy-handed approaches. State saving needs to capture the execution state of the logical processes in the forward time direction, which inevitably results in increased memory consumption; this problem is particularly acute when handling large-scale models. Reverse computation shifts the cost to the opposite time direction. In addition, for better efficiency, the modelers may need to supply handcrafted reversing functions that can further add to the obscurity of the model code.

Conservative synchronization requires explicit specification of causality in the model, so that logical processes can coordinate with one another to enable local event execution in strict timestamp ordering. From the appearance, the conservative approach burdens the modelers with the task of explicitly specifying the inter-dependencies of the logical processes as the lookahead, defined as the minimum amount of simulation time that one logical process can potentially alter the state of another process. In reality, such specification can be simplified as a model graph, which consists of logical processes connected by links with weights equal to the lookahead. For example, if the current simulation time at $\mathsf{LP_A}$ is $\mathsf{t_A}$ and there exists a channel from $\mathsf{LP_A}$ to $\mathsf{LP_B}$ with lookahead $\mathsf{x_{AB}}$, it means that in the future all simulation events sent from $\mathsf{LP_A}$ to $\mathsf{LP_B}$ are guaranteed to carry a timestamp no smaller than $(t_A + x_{AB})$. More sophisticated forms of lookahead (e.g., [Nic88, HN93]) can also be expressed in a similar fashion.

Albeit simplistic, what's important is this method offers a clear separation of concerns between the domain modelers and the parallel simulation experts. The graph is constructed based on the semantics of the specific model. For example, a network model can be a graph with nodes representing router, and links representing connections between the routers with weights indicating the minimum transmission delays. Once the graph is defined, parallel synchronization algorithms can be applied based on the topology of the graph, insomuch free from the modeling concerns.

3.2 Related Work

It is important that conservative algorithms must identify lookahead in the model: a positive lookahead suggests that an LP can operate within a certain period of time independent from other LPs. Lookahead thus implies the inherent asynchrony in the simulation model. Exploiting lookahead takes on two directions. One direction focuses on extracting lookahead from the model characteristics. Lookahead comes in different forms [Nic96]. For example, by pre-sampling job service times and branch destinations when a job is entering a first-come-first-serve (FCFS) queue, one can predict the earliest time a job will be sent to the subsequent queues [Nic88]. For another example, one can exploit mathematical property when simulating continuous-time Markov chains (CTMC) to determine the potential synchronization points between the LPs [HN93]. Obviously, lookahead extraction is model specific.

The other direction focuses on lookahead extrapolation for general applications. In particular, the model topology can play an important role in the lookahead computation. For example, Lubachevsky's Bounded Lag algorithm [Lub88] takes advantage of the minimum propagation delay between the LPs and the so-called opaque period during which the state of an LP is not affected by other LPs due to the model's non-preemptive behavior. The algorithm introduces a time interval, called the lag, *B*, using which the algorithm computes the "sphere of influence" encompassing all LPs that can possibly affect a given LP within *B* units of simulation time. Ayani's Distance Between Objects algorithm [Aya89] is another case in point. The algorithm exploits the distance between the LPs using shortest-path to determine the LP's LBTS. The composite synchronization falls in this category, which assumes that lookahead can be expressed through the model topology in the form of an LP graph.

3.2.1 Composite Synchronization

The conservative synchronization methods can be classified as either synchronous or asynchronous approaches. The original composite synchronization algorithm [NL02] considers the performance problem arising from the mismatch between the model topology and the synchronization scheme. The synchronous approaches (e.g., [Nic93, CS89, Lub88, Aya89]) exploit the computational efficiency of the collective operations, such as barriers and reductions, and are more suitable for densely connected models, where it is more likely that any LP may interact with (i.e., affect and be affected by) any other LP during the course of the simulation. The cost of the synchronous methods is directly related to the size of the synchronization window, which unfortunately is determined by the *worst-case* lookahead between the LPs. In contrast, the performance of the asynchronous approaches (e.g., [CM79, XUSC99, SS89, CT90]) does not easily get stuck on the worst-case scenario as they focus on the pair-wise interactions between the LPs. However, since the cost is closely related to the connection degree at each LP, the asynchronous methods are at a disadvantage for handling densely connected models.

The composite synchronization algorithm aims to combine the synchronous and asynchronous approaches in order to avoid the potential performance pitfalls from using either method alone. The composite algorithm works as follows. Consider a model topology represented as a graph, where the nodes are the LPs and the links are the channels between the LPs. The algorithm partitions the channels between the LPs as either synchronous or asynchronous channels. The size of the global synchronization window is then set to be the minimum latency among all synchronous channels. The algorithm runs a barrier synchronization among all LPs at the start of each synchronization window. Within a synchronization window, the algorithm uses the CMB algorithm to carry out the synchronization among the LPs in the subgraph that consists of only asynchronous channels. During the execution, all events that traverse the synchronous channels are temporarily stored at the sending processor. At the end of the synchronization window, these events are distributed and delivered to their destinations using a collective operation. Since these are the only events to be received by the LPs through the synchronous channels during the next window, the composite algorithm thus only needs to consider asynchronous channels before the next barrier synchronization.

The performance of the composite synchronization algorithm depends on the channel assignment. In [NL02], Nicol and Liu formulate the channel assignment problem as an optimization problem and show that the optimal policy has a threshold structure—one uses a threshold T to partition the channels: those with latencies greater than or equal to T are classified as synchronous channels, and the rest are classified as asynchronous. In practice, the simple threshold structure allows the algorithm to "search" for the optimal threshold by dynamically changing the threshold to seek the one that produces the best performance.

Nicol and Liu's composite algorithm was implemented only on shared-memory multiprocessors. Our approach extends the original algorithm for the distributed-memory multiprocessor and multicore environments, mostly common in today's HPC realm. Our extended algorithm makes a judicious distinction between synchronization over distributed memory and over shared memory. The extended algorithm is a hybrid approach as it combines two distinct synchronization strategies; it is also a hierarchical approach as it differentiates global distributed-memory synchronization and local shared-memory synchronization. The same philosophy has been adopted earlier. For example, the Local Time Warp approach [RAT93] combines a global conservative window-based synchronization with a local optimistic synchronization. For another example, the DaSSF simulator features a two-level synchronization scheme; it uses a barrier window-based algorithm for synchronizing distributed-memory machines, and an asynchronous method for synchronizing shared-memory multiprocessors [LN01].

3.3 Hierarchical Composite Synchronization

We start with a model graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$, where V is the set of LPs and E is the set of directed channels between the LPs. Let $l_{\mathsf{e(x,y)}}$ be the latency (i.e., the lookahead) of the channel $\mathsf{e(x,y)} \in \mathsf{E}$ connecting LP $\mathsf{v_x} \in \mathsf{V}$ to LP $\mathsf{v_y} \in \mathsf{V}$. We assume $l_{\mathsf{e(x,y)}} > 0$ for all channels. Suppose that the simulation is run on the target platform that consists of m distributed-memory machines, each having p_i processors or cores[1] that communicate over shared memory, where $0 \le i < m$.

We first run a graph partitioning algorithm that assigns each LP to a processor. Let $\mathsf{V_{ij}}$ be the set of LPs assigned to processor (or core) j on machine i, where $\bigcup \mathsf{V_{ij}} = \mathsf{V}$ and $\bigcap \mathsf{V_{ij}} = \phi$ for $0 \le i < m$ and $0 \le j < p_i$. Let $\mathsf{E_G}$ be the set of channels that span across different distributed-memory machines. That is, $\mathsf{E_G} = \{\mathsf{e(x,y)} \in \mathsf{E} \mid \mathsf{v_x} \in \mathsf{V_{ij}} \wedge \mathsf{v_y} \in \mathsf{V_{kl}} \wedge i \ne k\}$. We use $\mathsf{V_i}$ to denote the set of LPs on machine i, that is, $\mathsf{V_i} = \bigcup_{j=0}^{p_i-1} \mathsf{V_{ij}}$, and we use $\mathsf{E_i}$ to denote the set of channels between LPs on the same machine i, that is, $\mathsf{E_i} = \{\mathsf{e(x,y)} \in \mathsf{E} \mid \mathsf{v_x} \in \mathsf{V_i} \wedge \mathsf{v_y} \in \mathsf{V_i}\}$. In practice, we use a graph partitioner to divide

[1]Here, we do not distinguish between processors and cores within the processors, as much as shared memory is concerned. The algorithm nevertheless can be extended to handle more than two synchronization levels.

the LPs (each regarded as a unit of workload) among the processors and minimize the total link weights exposed between the partitions. This can be achieved by making link weights inversely proportional to the link latencies and using a graph partitioner, like METIS [KK98], to find an optimal partition with minimum cut. Furthermore, if one can predict the communication intensity between LPs, we can apply Liu and Chien's three-stage graph partitioning strategy [LC03], which takes into account both link delay and communication overhead. The algorithm runs a graph partitioner in the first time with the goal of finding a partition that maximizes the link latency, and in the second time for finding a partition that minimizes the communication intensity between the partitions. Finally, it creates another graph with new edge weights calculated from the results of two previous partitioning runs, and partitions the graph for the third time to obtain the final result.

Once we partitioned the model graph, we continue to classify the channels in the graph as either synchronous or asynchronous. We defer the description of our multi-threshold method to classify the channels in detail to section 3.4. Regardless of the method we use, we divide the channels into the set of synchronous and asynchronous channels. We denote $\mathsf{E}_{\mathsf{G}}^{\mathsf{S}}$ to be the set of synchronous channels and $\mathsf{E}_{\mathsf{G}}^{\mathsf{A}}$ to be the set of asynchronous channels that span across different distributed-memory machines. Similarly, we denote $\mathsf{E}_{\mathsf{i}}^{\mathsf{S}}$ to be the set of synchronous channels and $\mathsf{E}_{\mathsf{i}}^{\mathsf{A}}$ to be the set of asynchronous channels between LPs on machine i, where $0 \leq i < m$. We calculate the size of the global synchronization window, δ_{G}, as the minimum latency among the synchronous channels that span across the machines, that is, $\delta_{\mathsf{G}} = \min_{\mathsf{e(x,y)} \in \mathsf{E}_{\mathsf{G}}^{\mathsf{S}}} \{l_{\mathsf{e(x,y)}}\}$. Similarly, we calculate the synchronization window on each machine, δ_{i}, to be the minimum latency among the synchronous channels on the particular machine, that is, $\delta_{\mathsf{i}} = \min_{\mathsf{e(x,y)} \in \mathsf{E}_{\mathsf{i}}^{\mathsf{S}}} \{l_{\mathsf{e(x,y)}}\}$.

The composite algorithm runs a local barrier synchronization every δ_{i} units of simulation time among the processors on each machine i, and a global barrier synchronization every δ_{G} units of simulation time among all distributed-memory machines. Between these barriers, the composite algorithm runs an asynchronous algorithm on the subgraph, $\mathsf{G}^{\mathsf{A}} = (\mathsf{V}, \mathsf{E}^{\mathsf{A}})$, consisting of only asynchronous channels: $\mathsf{E}^{\mathsf{A}} = (\bigcup_{0 \le i < m} \mathsf{E}_{\mathsf{i}}^{\mathsf{A}}) \bigcup \mathsf{E}_{\mathsf{G}}^{\mathsf{A}}$. For each LP $\mathsf{v_x} \in \mathsf{V}$, we denote $\mathsf{I_x}$ to be the set of asynchronous channels ending at LP $\mathsf{v_x}$, and $\mathsf{O_x}$ to be the set of asynchronous channels starting from LP $\mathsf{v_x}$. The pseudo code of the hierarchical composite algorithm is described in Alg. 1; the algorithm is expected to run on each machine i and at each processor or core j.

Each processor or core maintains a priority queue, $\mathsf{Q_{ready}}$, containing all ready LPs sorted by their local simulation clock. We say an LP $\mathsf{v_x}$ is ready to run when the local simulation clock, $\mathsf{t_x}$, is smaller than its LBTS, calculated as the minimum time among of all incoming channels of $\mathsf{v_x}$, that is, $\min_{\mathsf{e} \in \mathsf{I_x}}\{\mathsf{t_e}\}$. Initially the queue is empty (line 1). Each processor also defines two sets, $\mathsf{EVT_G}$ and $\mathsf{EVT_L}$, to store the events traversing the global synchronous channels, $\mathsf{E}_{\mathsf{G}}^{\mathsf{A}}$, and those traversing the local synchronous channels on the same machine, $\mathsf{E}_{\mathsf{i}}^{\mathsf{A}}$, respectively. Initially the two sets are also set to be empty (line 1). We initialize the local simulation clock of all local LPs to zero and set the time of all incoming channels to the LPs to be the same as the latencies (lines 2-4).

We use t to indicate the start time of the current synchronization window, which starts from 0 (line 5). We use $\mathsf{w_G}$ to keep track of the next global synchronization point (among all distributed-memory machines), which is incremented each time by δ_{G} (lines 5 and 35). We use $\mathsf{w_L}$ to keep track of the next local synchronization point (among all processors or cores on the same machine), which is incremented each time by δ_{i} (lines 5 and 39). We use w to denote the end time of the current synchronization window, which is defined as the the smaller of $\mathsf{w_G}$ and $\mathsf{w_L}$ (lines 5 and 42). The algorithm thus iterates through the synchronization windows until the simulation time t reaches the simulation termination time $\mathsf{T_{term}}$ (line 6).

At the start of the synchronization window, all LPs are inserted into the ready queue (line 7). We use the variable done to keep track of the number of LPs that have already reached the end of the current synchronization window. The while-loop between line 9 and line 32 is used to schedule the LPs asynchronously until all of them reach the end of the synchronization window. In the while-loop between line 10 and line 27, an LP v_x with the earliest simulation clock is removed from the ready queue (line 11) and gets to run. The algorithm calculates its LBTS to be the minimum time of all incoming channels and the end time of the current synchronization window (line 12). It can then safely process all simulation events earlier than LBTS on LP v_x's event list (line 13). Processing events may generate more events on the same LP, in which case they are inserted into the LP's event list directly. The LP may also send events to other LPs via channels. If the events are sent through synchronous channels, they are put into EVT_G if they traverse global synchronous channels, and EVT_L if they traverse local synchronous channels. If the events are sent through asynchronous channels, they are delivered asynchronously. We discuss the implementation details of the event delivery mechanism in the next section.

Once all safe events are processed, the simulation clock t_x is updated to be LBTS (line 14). The algorithm makes a scan through all outgoing channels of LP v_x and updates the channel times (lines 15 and 16). If the subsequent LP v_y is on the same processor and it is blocked, we need to reinsert the LP onto the ready queue when the updated channel time advances its LBTS (lines 17-21). If v_y is on a different processor or a different machine, we send a null message (line 23). When the algorithm exhausts all ready LPs and if there are still LPs not reaching the end time of the synchronization window, the algorithm will wait for incoming null messages to update the channel times and LBTS of some LPs, so that they become ready to run (lines 28-31).

At the end of the synchronization window, if it happens to be a global synchronization point, all machines enters a global barrier and perform an all-to-all exchange of events stored in $\mathsf{EVT_G}$ (lines 34 and 36). If the end of the synchronization window is a local synchronization point, all processors enter a local barrier and perform an all-to-all exchange of events stored in $\mathsf{EVT_L}$ (lines 38 and 40). The exchange of events depends on the implementation. In the next section, we describe our implementation in the *MiniSSF* simulator.

3.4 Performance Model of the Algorithm

The performance of the hierarchical composite synchronization algorithm depends heavily on how the channels are classified. The global synchronization can be either a pure barrier-based algorithm, if all cross-machine channels are classified as synchronous channels, or a pure asynchronous CMB-like protocol, if the all channels are all qualified as asynchronous. Everything in-between is also a possibility. Furthermore, the choice of the global synchronization can be made independent from that of the local synchronization. In [NL02], Nicol and Liu found that the optimal channel classification (for one level) can be obtained using a threshold. In this section, we develop a simple cost model for the hierarchical composite algorithm, and present a method of applying linear regression to calculate the necessary parameters for the target parallel platform. We can then use the parameters to help search for the thresholds for the optimal channel classification.

We suppose that the hierarchical composite synchronization algorithm is running for the model graph G on the parallel platform that consists of m machines, with p_i processors on machine i ($0 \leq m < m$). We use $\mathsf{C_{global}}$ to denote the cost at the global synchronization among the distributed-memory machines, $\mathsf{C^{i}_{local}}$ to denote the cost at the local synchronization between processors at machine i, and $\mathsf{C^{ij}_{async}}$ to denote the cost of running the asynchronous algorithm on machine i and processor j. The running time of the hierarchical composite algorithm can be calculated as follows:

$$\mathsf{C}_{\mathsf{global}} + \max_{0 \le i < \mathsf{m}} \{\mathsf{C}^{\mathsf{i}}_{\mathsf{local}} + \max_{0 \le j < \mathsf{p_i}} \mathsf{C}^{\mathsf{ij}}_{\mathsf{async}}\} \tag{3.1}$$

Here, we use the maximum to account for the possible load imbalance among the shared-memory processors at a machine, and among the distributed-memory machines.

The cost at the global synchronization consists of the cost of the barriers and the cost of distributing events sent through the synchronous channels spanning between the distributed memory machines. The cost of the barriers is inversely proportional to the global synchronization window size, δ_{G}. The cost of distributing the events can be approximated as proportional to the number of synchronous channels spanning across different machines, $|\mathsf{E}^{\mathsf{S}}_{\mathsf{G}}|$. That is,

$$\mathsf{C}_{\mathsf{global}} = \frac{\mathsf{c}_1}{\delta_{\mathsf{G}}} + \mathsf{c}_2|\mathsf{E}^{\mathsf{S}}_{\mathsf{G}}| \tag{3.2}$$

where c_1 and c_2 are the proportional constants. The cost at the local synchronization can be expressed similarly:

$$\mathsf{C}^{\mathsf{i}}_{\mathsf{local}} = \frac{\mathsf{c}_3}{\delta_{\mathsf{i}}} + \sigma_4|\mathsf{E}^{\mathsf{S}}_{\mathsf{i}}| \tag{3.3}$$

where c_3 and σ_4 are also the proportional constants. We use the same constants, c_3 and σ_4, for all machines assuming the machines are homogenous.

The cost for running the asynchronous algorithm consists of the cost of processing the events of all LPs on machine *i* and processor *j*, and the cost of asynchronously scheduling the LPs. The frequency of scheduling an LP can be approximated as inversely proportional to the length of the shortest cycle through the LP in the model graph. The cost of each time scheduling an LP to run is proportional to the number of incoming asynchronous channels (for computing LBTS and for retrieving events from the channels' mailboxes), and the number of outgoing asynchronous channels (for sending events and for updating the channel times).

We let π_x be the cost for processing all events on LP v_x. We use τ_x to denote the shortest cycle length through LP v_x. We define I_x as the set of incoming asynchronous channels to v_x, O_x^P as the set of outgoing asynchronous channels connecting to other LPs on the same processor, O_x^M as the set of outgoing asynchronous channels connecting to other LPs on different processors and yet on the same machine, and O_x^G as the set of outgoing asynchronous channels spanning across different machines. Here we make a distinction among the different types of outgoing channels because they incur different costs. The overall cost of running the asynchronous algorithm on machine *i* and processor *j* can thus be expressed as:

$$\begin{aligned} C_{async}^{ij} &= \sum_{v_x \in V_{ij}} (\pi_x + \frac{\sigma_5|I_x| + \sigma_6|O_x^P| + \sigma_7|O_x^M| + \sigma_8|O_x^G|}{\tau_x}) \\ &= \mathscr{A}_{V_{ij}} + \sigma_5 \mathscr{I}_{V_{ij}} + \sigma_6 \mathscr{O}_{V_{ij}}^P + \sigma_7 \mathscr{O}_{V_{ij}}^M + \sigma_8 \mathscr{O}_{V_{ij}}^G \end{aligned} \tag{3.4}$$

where $\mathscr{A}_{V_{ij}} = \sum_{v_x \in V_{ij}} \pi_x$ is the cost for processing events for all LPs residing on the processor (that is, V_{ij}), $\mathscr{I}_{V_{ij}} = \sum_{v_x \in V_{ij}} |I_x|/\tau_x$ is the cost of scanning the incoming asynchronous channels of LPs in V_{ij}, and $\mathscr{O}_{V_{ij}}^P = \sum_{v_x \in V_{ij}} |O_x^P|/\tau_x$, $\mathscr{O}_{V_{ij}}^M = \sum_{v_x \in V_{ij}} |O_x^M|/\tau_x$, and $\mathscr{O}_{V_{ij}}^G = \sum_{v_x \in V_{ij}} |O_x^G|/\tau_x$ represent the cost of scanning the outgoing asynchronous channels on the same processor, across processors within the same machine, and across different machines, respectively. We use the same proportional constants, σ_5 to σ_8, for different processors and machines, assuming the parallel platform is homogenous.

The performance model for the composite algorithm expressed in Equations (3.1) to (3.4) can be of some theoretical value. However, in practice, in order for us to be able to estimate the runtime we need to make further simplifications. We add three more assumptions. First, we assume that the model is perfectly balanced. That is, the model can evenly distribute the workload among the LPs. This is normally true for most large-scale simulation scenarios, which usually consist of a large number of similar entities generating a large number of simulation events. Second, we assume that the simulation is running with a perfect balance on a set of homogenous machines each with *p* processors. This is also true in general if we use a good graph partitioning algorithm and if the simulation workload does not change significantly throughout the simulation. Third, we assume that the composite synchronization algorithm adopts the same local synchronization window size, δ_L, on all distributed-memory machines. This is true if the latencies for the cross-processor synchronous channels (E_i^S) have the same distribution on all machines.

With these assumptions, the cost of the hierarchical composite algorithm can be simplified as a function of δ_G and δ_L shown below:

$$\begin{aligned}
& C_{global} + \frac{1}{m}\sum_{0\le i<m} C^i_{local} + \frac{1}{mp}\sum_{\substack{0\le i<m\\0\le j<p}} C^{ij}_{async} \\
&= \left(\frac{c_1}{\delta_G} + c_2|E_G^S|\right) + \left(\frac{c_3}{\delta_L} + \frac{\sigma_4}{m}\sum_{0\le i<m}|E_i^S|\right) + \\
&\quad \frac{1}{mp}\left(\mathscr{A}_V + \sigma_5\mathscr{I}_V + \sigma_6\mathscr{O}_V^P + \sigma_7\mathscr{O}_V^M + \sigma_8\mathscr{O}_V^G\right) \\
&= c_0 + \frac{c_1}{\delta_G} + c_2|E_G^S| + \frac{c_3}{\delta_L} + c_4\sum_{0\le i<m}|E_i^S| + \\
&\quad c_5\mathscr{I}_V + c_6\mathscr{O}_V^P + c_7\mathscr{O}_V^M + c_8\mathscr{O}_V^G
\end{aligned} \tag{3.5}$$

where $\mathscr{A}_V = \sum_{v_x\in V}\pi_x$ is the cost for processing events for all LPs, $\mathscr{I}_V = \sum_{v_x\in V}|I_x|/\tau_x$ is the cost of scanning the incoming asynchronous channels of all LPs, $\mathscr{O}_V^P = \sum_{v_x\in V}|O_x^P|/\tau_x$, $\mathscr{O}_V^M = \sum_{v_x\in V}|O_x^M|/\tau_x$, and $\mathscr{O}_V^G = \sum_{v_x\in V}|O_x^G|/\tau_x$ are the costs of scanning the outgoing asynchronous channels on the same processor, across processors within the same machine, and across different machines, for all LPs. We set $c_0 = \mathscr{A}_V/mp$, $c_4 = \sigma_4/m$, and $c_5 = \sigma_5/mp, \cdots, c_8 = \sigma_8/mp$.

In practice, we can use the following algorithm to search for the optimal combination of δ_G^* and δ_L^* for the best performance on a target platform:

1. We run a graph partitioning algorithm to partition the model graph *G* for the target platform with *m* machines each with *p* processors. This can be achieved by first

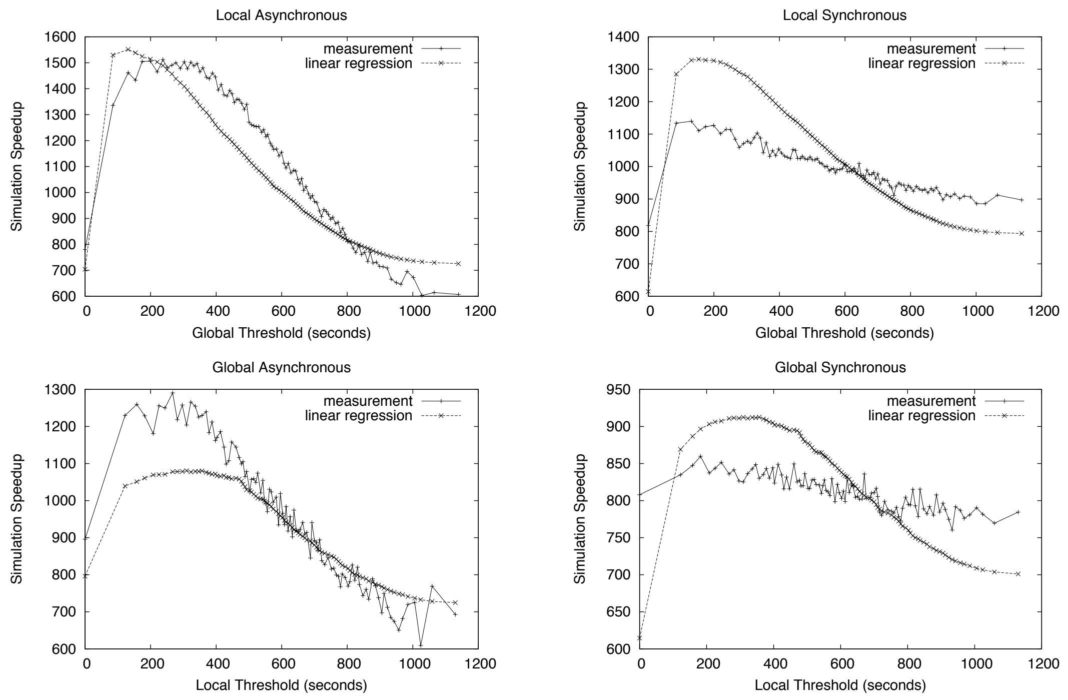


Figure 3.1: Cost estimation using linear regression.

partitioning the graph among the machines and then individually partitioning the subgraph at each machine among the processors.

2. We make a few pilot runs of the composite algorithm with different combination of δ_G and δ_L, and measure the runtime. We can then use linear regression to estimate the constants, c_0 to c_8. Once we have the constants, we can run the search offline. Since the global synchronization and local synchronization are independent, we can separately search for δ_G^* and δ_L^* in the steps to follow.

3. We try out different values of δ_G using all distinct latencies of channels spanning across distributed-memory machines, i.e., in E_G^S. We find the optimal δ_G^* that minimizes:

$$\frac{c_1}{\delta_G} + c_2|E_G^S|$$

4. Similarly, we try out different values of δ_L using all distinct latencies of channels spanning across shared-memory processors on the same machine, i.e., in $\bigcup_{0\leq i<m} E_i^S$.

We find the optimal δ_L^* that minimizes:

$$\frac{c_3}{\delta_L} + c_4 \sum_{0 \le i < m} |E_i^S| + c_5 \mathscr{I}_V + c_6 \mathscr{O}_V^P + c_7 \mathscr{O}_V^M + c_8 \mathscr{O}_V^G$$

In steps 3 and 4, if the channels have a large number of distinct latencies, we may opt to take only a sample of the distinct latencies to speed up the search. Furthermore, if we can assume that the behavior of the model is consistent across different model sizes, it is possible to create a performance model that allows us to run measurements using a smaller model on a subset of machines, and then project the results for large models running on the full-size target parallel machine. We leave this option for future work.

3.5 Preliminary Experiments of the Algorithm

To demonstrate the effectiveness of our approach, we conducted an experiment using a queuing network model. We used BRITE [MLMB01] to generate a network topology of 9,600 nodes randomly connected using a probability model. In our case, we chose the router-level Barabási-Albert model that generates the topology with a power-law distribution in the frequency of node degrees. BRITE calculates the link delays according to the placement of the nodes: it first places the nodes in a 2D plane uniformly at random and then calculates the link delays between the nodes to be proportional to the Euclidean distances. In our case, we observed the link delays range widely between 8 to 1,350 milliseconds. For the queuing model, we set the service time at each queue to be exponentially distributed with a mean of one second. After service, the job departs from the current queue and randomly joins one of the connected queues after experiencing the link delay between the two queues. At the start, each queue has an average of 100 jobs, sampled from a Poisson distribution.

We ran the experiment on a cluster of eight machines connected by a gigabit Ethernet switch. Each machine is equipped with two hex-core 2.4 GHz AMD Opteron CPUs and

32 GB of RAM. We used METIS [MET] to partition the model in three rounds: it first partitioned the entire network among the eight machines, then the nodes that belong to each machine were partitioned among the twelve cores, and finally the nodes that belong to each core were partitioned among ten logical processes. In the end, we had a model with 960 logical processes.

To calculate the parameters of our performance model, we used 100 different global threshold values sampled from the delays of the links spanning across the machines. We measured the runtime using these thresholds for global synchronization, while setting the local synchronization to be either purely synchronous or purely asynchronous. Similarly, we used another 100 different local threshold values sampled from the delays of the links spanning across processors on the same machine. We measured the runtime using these thresholds for local synchronization and setting the global synchronization to be either purely asynchronous or purely synchronous. Afterwards, we used MATLAB to calculate the nine parameters (c_0 to c_8) using linear regression. After getting the performance model, we estimate the global and local thresholds that can achieve the best performance.

Fig. 3.1 shows the simulation speedup (i.e., the ratio of the parallel execution time over the simulated time). The top two plots show the results as we vary the global synchronization threshold (δ_G). The top left plot is from using pure asynchronous null-message-based approach for synchronizing the LPs located on the same machine, and the top right is from using pure synchronous window-based synchronization for the LPs located on the same machine. The bottom two plots show the results as we vary the local synchronization threshold (δ_L), while keeping the global synchronization to be purely asynchronous and purely synchronous methods, respectively. Overall, our model can track the relative changes in the performance of the composite synchronization approach as we vary the thresholds. The model, however, is not indicative to the absolute costs, and as such still requires further tuning in order to increase the accuracy in the performance estimation.

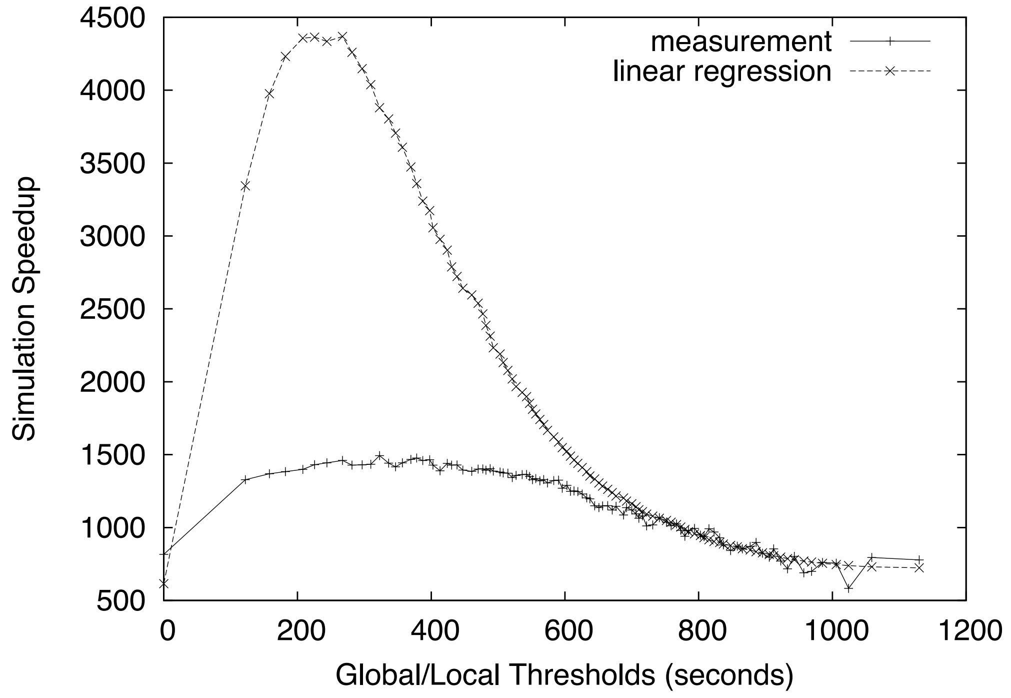


Figure 3.2: Cost prediction.

Fig. 3.2 shows the measured runtime compared to the model predicted performance through linear regression, as we simultaneously increase the global and local thresholds. Again, the model is less accurate in predicting the absolute cost of the composite algorithm; but it can nevertheless predict the relative performance in accordance with the changing thresholds. We suspect some of the inaccuracies come from the perfect load balancing assumption we made in the model. We are currently investigating the causes.

Fig. 3.3 shows the speedup of the composite algorithm using the best combination of global and local thresholds (237 and 267 in this case) over pure synchronous and asynchronous approaches used separately for synchronizing the global distributed-memory machines and for synchronizing the local shared-memory multiprocessor multicore machines. In particular, the hierarchical composite algorithm achieved a speedup of as much as 4.8 over the globally asynchronous and local asynchronous approach, which is the traditional CMB-style algorithm, on the 8-node 96-core cluster. Our algorithm also yielded

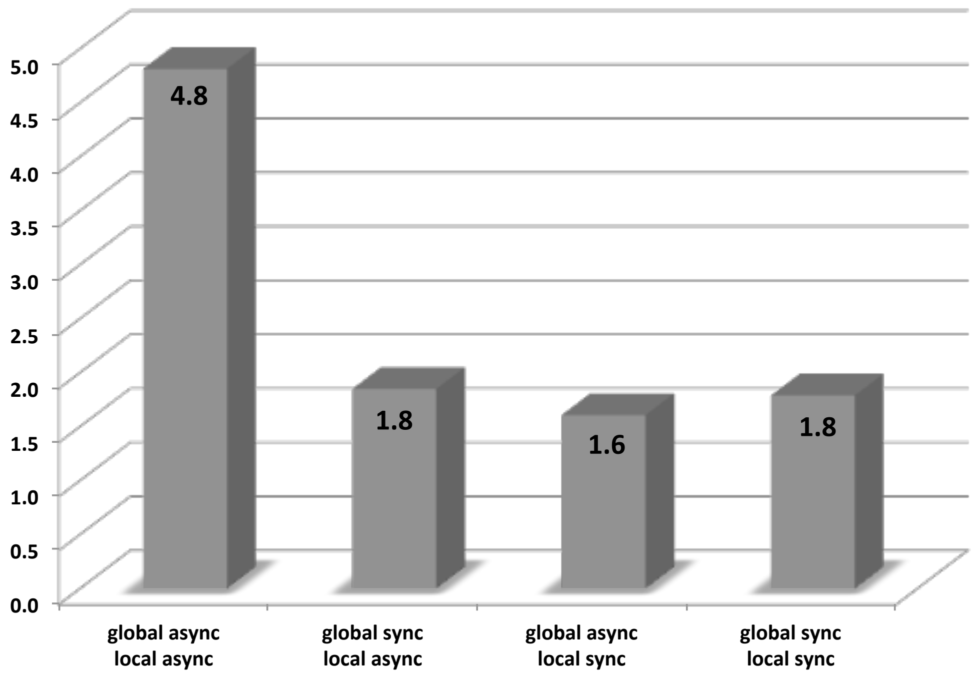


Figure 3.3: Optimal speedup over pure sync and async methods.

a performance improvement of 1.8 over the synchronous approach used at both levels, which is the traditional window-based protocol, like YAWNS. We also observed the hierarchical composite algorithm obtained a speedup of 1.7 over the two-level synchronization scheme that uses the composite synchronization for local synchronization, and uses the window-based approach for global synchronization, as was the implementation of the original composite algorithm.

3.6 The Simulator Implements the Synchronization—MiniSSF

We implemented the hierarchical composite synchronization algorithm in the new *MiniSSF* simulator, which implements a core subset of the Scalable Simulation Framework (SSF) API [CNO99], for running large complex models for distributed-memory multiprocessor multicore platforms.

3.6.1 Introduction

The scale and complexity of modern computer systems have grown rapidly. It is thus vitally important to provide an expressive and flexible simulator capable of handling models of different sizes, at different modeling details, and with different levels of complexity. A parallel discrete-event simulator is capable for users to easily develop large complex models, and simultaneously offer transparent scalability and performance when running the models on high-end computing platforms. There have been many parallel discrete-event simulators shown to be able to run large-scale models (e.g., [BMT$^+$98, CBP00b, CNO99, DFP$^+$94, Per]). Recent performance studies on the parallel simulation on supercomputers have also shown encouraging results (e.g., [BCH09, CP10, FPP$^+$03, Per07]).

Scalable Simulation Framework (SSF) is an application programming interface (API) designed for developing parallel simulation models [CNO99]. It is based on modular design through which potential parallelism in the model can be identified and exploited. Several salient features of SSF have made it quite attractive as a general parallel simulation API. First, a model is described in SSF simply as a connected graph; the detailed logic of the model is hidden inside the specific entities. This allows building large complex models in a modular fashion. Second, SSF provides a process-oriented simulation view, where a model can be expressed naturally as a collection of interacting processes. Finally, an SSF model is largely independent of the synchronization protocol and the parallel platform. This independency provides a clean separation of concerns between the domain modelers and the developers of the parallel simulator.

Deficiencies of SSF

The first C++ implementation of SSF is started in 1998. The simulator has gone through several rounds of careful performance analysis and tuning [LNPP99], and has demonstrated capable of simulating large-scale infrastructure networks [CLL+99]. Since then, the simulator and its variants have been used in many other applications, including simulations of parallel computers, interconnection networks, file systems, cellular networks, wireless ad hoc networks, sensor networks, power grids, etc. Despite the simulator's widespread use, our experience with the development and maintenance of the simulator has brought on the realization of several deficiencies in the original design.

First, the original SSF API contains many powerful features, such as dynamic entity creation and deletion, dynamic mapping of communication channels between entities, entity realignment, as well as the capability pausing and resuming simulation during execution. We observe that these features are seldom used in model development; and yet they add significant complexities both in the implementation and maintenance of the simulator.

Second, the SSF API does not provide a standard way of creating and initializing models on distributed-memory machines. Distributed-memory machines are common parallel platforms of today; most supercomputers currently available consist of distributed-memory machines with shared-memory multiprocessors and multicores. The lack of an intuitive interface for instantiating models on these platforms is a serious inhibitor for further expanding its use.

Third, the SSF implementation depends on an efficient user-space multithreading mechanism to support the process-oriented simulation view. User-space multithreading requires source-code transformation, which unfortunately has not been fully automated in the original implementation. As a result, the user is burdened with the task of having to properly annotate the source code. This adds significant complexity to the model development and for debugging.

Fourth, the parallel simulation synchronization algorithm implemented in the SSF implementation has been developed and optimized only for shared-memory multiprocessor machines [NL02]. A less optimal barrier-based algorithm is used for synchronizing the distributed-memory machines [LN01]. The synchronous approach may result in sub-optimal performance if the barrier synchronization window is too small for certain models.

3.6.2 MiniSSF

We revisit the SSF design and describe a new implementation of the parallel simulator, called *MiniSSF*, which improves upon the previous implementation. The new simulator takes on the minimalistic approach originated from the SSF design, and we get rid of some of the superfluous features in SSF that are not commonly used by modelers. This also removes the unnecessary obscurity of the model code for analysis and transformation. We include a standard API for creating, initializing, and running the simulation models on distributed-memory platforms, following the well-known single-program-multiple-data (SPMD) programming paradigm. Using the same intuitive interface, the simulator is able to run either sequentially on a single machine, or in parallel on a compute cluster of machines with multiple processors and cores.

We provide a fully automated source-code analysis and transformation tool for efficient user-space multithreading to support the process-oriented simulation view. Furthermore, we implement a hierarchical composite synchronization protocol that can automatically tune its performance based on the model and the underlying parallel platform. The new simulator is itself a multi-threaded parallel program, using only pthreads and MPI, both commonly available on today's high-end computing platforms. This makes the simulator extremely portable across different parallel platforms we have so far encountered.

Programming Interface of MiniSSF

It is important that a simulator provides the necessary software constructs for the user to easily build large complex models that can run efficiently on today's parallel platforms. Our simulator largely follows the original SSF API, however, with some important changes to ease model development. In this section, we describe the *MiniSSF* API focusing on the specific design issues and extensions. SSF defines five core constructs: entity, process, in-channel, out-channel, and event.

An *entity* is a container for state variables collectively representing a component in the target system. For example, we can use an entity to model a queue in the queuing network. The entities are connected by mapping the *out-channels* of the entities with the *in-channels* of other entities. The channels are the communication end-points between the entities with specific delays. *Events* are messages sent through the channels. An out-channel can be connected to multiple in-channels, in which case an event written to the out-channel will be delivered by the simulator to all mapped in-channels with the specific delays. Similarly, an in-channel can be mapped from multiple out-channels so that it can receive events from multiple sources. Within each entity, one can create *processes* to perform simulation activities. A process can be blocked waiting for events to arrive on the entity's in-channels, or for a certain duration of simulation time to pass.

SSF exemplifies the common task/channel model commonly used for developing parallel applications [Fos95]. A parallel application is defined as set of independent tasks which interact by sending and receiving messages via channels. In doing so, it avoids the use of global shared data and thus can be mapped easily onto a parallel platform. The computation is divided into separate tasks that can run in parallel, whereas the data dependencies among the tasked can be easily identified through the communication channels. In SSF, each task is represented as an entity defined with a set of local state variables. An entity can have a set of in-channels and out-channels as I/O ports to communicate with other entities. An SSF model consists of entities interconnected via the channels (it's a graph). Within an entity, one or more processes can be defined to describe the specific logic of the corresponding task/component of the target system. A process describes the changes of the entity's state according to time or in response to the messages sent from the other entities.

Process-oriented Simulation

SSF offers a process-oriented simulation view for the user to conveniently describe the model as a set of interacting simulation processes. Each process is an independent flow of control that specifies the state transition of the logical component represented by its owner entity. Implementing process-oriented view requires multithreading: a simulation process must be able to suspend its execution in the middle of a function. Also, the simulation engine needs to be designed seamlessly with the multithreading support: dispatching processes must be a core function of the event processing mechanism. To do multithreading, one can employ an existing thread package, such as pthreads.

A main drawback of this approach is the overhead. A full-fledged thread implementation requires each thread maintain its own stack and registers, as well as necessary bookkeeping information, such as scheduling properties and signaling mechanisms. Furthermore, although the threads are usually implemented as light-weight processes, they still incur non-trivial overhead during context switches. Both problems would be acute especially for large-scale models, which typically would consist of a huge number of simulation processes (which warrants the use of parallel simulation).

To avoid this problem, *MiniSSF* follows the design of its predecessor, using its own lightweight threading, via source-to-source translation and dependence on the simulation modeler using the right pragmas to suspend and resume the thread execution. The advantage of this approach is that the space consumption is now tailored to only what is explicitly used by each simulation process. The thread scheduling and context switching can be carefully orchestrated as part of the simulation event processing to reduce context switch overhead.

To allow a simulation process to suspend and resume execution, each simulation process needs to maintain a call chain, which keeps a snapshot of the sequence of the function calls together with their local variables, so that they can be restored after resuming execution. Since a simulation process can only be suspended as a result of executing a wait statement in *MiniSSF*, it is relatively easy to identify the call chain leading to a process suspension. We define a procedure as a function that either contains a wait statement or calls an other procedure. A call chain is a sequence of procedures from the starting procedure to the one that contains the wait statement that causes the process suspension.

The call chain of each simulation process is represented in *MiniSSF* as a linked list allocated from the program heap. Whenever a procedure is called, a procedure record is created and added to the head of the linked list (i.e., at the top of the stack). Each procedure record contains a pointer to the corresponding function, a copy of the local variables defined in the function, and a program counter, an integer indicating the next instruction to be executed once the function gets to be executed. When the simulation process resumes execution from the wait statement, the simulator will call the function indicated by the procedure record at the head of the call chain. A jump table inserted at the beginning of the function will direct the control to the specific instruction after the wait statement. When the function returns, the simulator will remove the procedure record from the head of the linked list and call the function indicated by the next procedure record (which now becomes the top of the stack), all until the control gets back to the starting procedure.

Automated Source Code Translation

To support hand-crafted multithreading, we need to perform source-code analysis to identify those functions where we embed necessary instructions to support thread suspension and resumption. Source-code analysis requires syntactic information, which can be obtained from a compiler. In the absence of a general full-fledged C++ parser that one could easily perform source-code analysis and transformation, the original implementation of SSF offers instead a source-to-source translator, written in Perl, to perform simple text-based code instrumentation. It's error-prone and significantly adds to the model complexity.

Acutely aware of these problems caused by the manual source-to-source translation, we introduce a fully automated source-code analysis and transformation mechanism for *MiniSSF*. For C++ compilation, we use *clang*, which is a C language family frontend for the LLVM compiler [cla]. Clang provides good support for static code analysis and source-to-source transformation, allowing users to add plug-in code to analyze and manipulate the abstract syntax tree (AST) obtained from parsing the source code.

We use clang to analyze the source code and identify several program features. First, we identify all procedures in the program. As mentioned earlier, procedures are functions that call the wait statements or other procedures. Second, we identify all places inside each procedure function that perform procedure calls. Third, we identify all places inside each procedure function where the function may return. Fourth, we identify the definition of all local variables at each procedure function, which also include function parameters and variables with different scopes within the function body. Finally, we identify all references to the local variables within the procedure function.

After that, we use clang to perform source-code transformation. First, for each procedure, we create a procedure record type, which includes a pointer to the procedure function, a set of variables corresponding to the function's local variables, and a program counter, an integer entry code indicating the instruction at which the function should start once the execution of the function resumes. Second, we replace the references to the local variables inside the procedure function with those to the corresponding variables defined in the procedure record. In this way, all local variables will be kept in the call chain located in the program heap rather than on the stack. As such, they can persist across the process suspension. Third, at the beginning of each procedure function, we insert a jump table. We add a label with a unique entry code at the first instruction following each procedure call or wait statement. The jump table is basically a switch statement, which directs the control (using `goto`) to the specific label corresponding to the entry code specified in the procedure record. Finally, at the place of each procedure call, we add code, which creates the callee's procedure record and adds it to the head of the call chain. Similarly, at each place the procedure returns, we add code, which removes the procedure record from the call chain and returns the control back to the simulator.

Our implementation consists of three modules. The first module is a clang plug-in, which is a dynamic library loaded by clang at run time. The first module will be run for each source file to perform the code analysis. More specifically, the module collects the information about the class hierarchy defined in the user's source code, retrieves the prototype of the methods defined in each class, and identifies all method calls inside the body of each method definition. After the first module is run on all source files, we run the second module to gather the information and create a call graph that represents what method may call what other methods. We can then identify the procedures and procedure calls by making a depth-first traversal of the call graph from the starting procedures. The third module is also a clang plug-in and is used again to parse each source file. At this time, it can perform the needed source-code transformation on the procedures identified by the first two modules.

3.6.3 The Implementation of Hierarchical Composite Synchronization

We implement the hierarchical composite synchronization algorithm [LR12] in *MiniSSF*. The algorithm has already been elaborated in section 3.3, so we focus on the implementation here. In particular, we some arrangements toward the issues related to running the algorithm on today's common parallel computing platforms.

Overall Implementation

The user runs the simulation on the target platform with one or more distributed-memory machines, each with several processors or cores. On each machine, the simulation starts from the main function, in which the user creates the entities along with the processes and channels, and connects the entities by mapping the out-channels with the in-channels. During the initialization, the user can also specify temporal alignment of the entities: co-aligned entities share the same timeline and advance in simulation time synchronously; that is, they can access each other's state variables without having to send and receive events through the channels. Once the initialization completes, the user basically creates the model graph consisting of LPs, which represent the co-aligned entities, and channels between the LPs, aggregated from the mapping from out-channels to in-channels.

The simulator is a multi-threaded MPI program. A thread is created on each processor and all LPs created for the machine are partitioned among the processors. The simulator also creates two additional threads—a reader thread and a writer thread—to handle communications with other distributed-memory machines. We use a facility, called mailbox, for communication between the threads. A mailbox consists of a linked list of events, a mutex to prevent simultaneous access by multiple threads, and a conditional variable for signaling between the data producer and the data consumer. We have three types of mailboxes in the simulation. The writer thread maintains a "remote mailbox" to store events

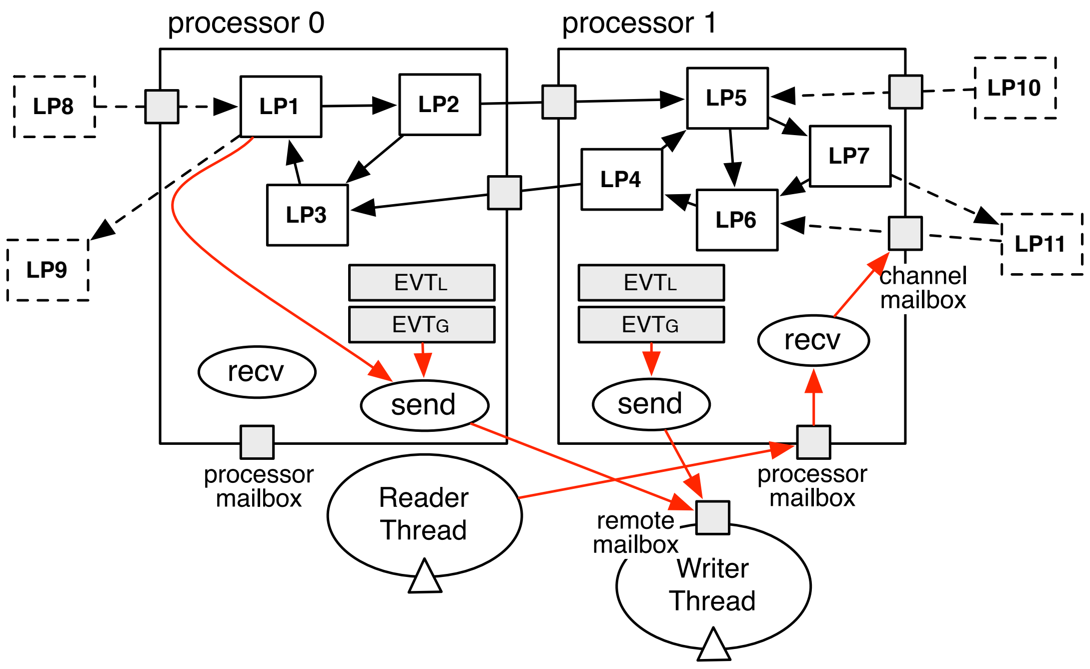


Figure 3.4: Event delivery mechanism.

from local processors waiting to be sent to different machines. Each processor also maintains a "processor mailbox" to store the events given to it by the reader thread received from other machines. For each channel connecting LPs belonging to different processors or different machines, the receiving LP maintains a "channel mailbox" to store the events delivered to it from the sending LP, before the events are inserted into the receiving LP's event list.

Fig. 3.4 illustrates the event delivery mechanism using an example with seven LPs (LP_1 to LP_7) divided between two processors; they are also connected with other LPs instantiated on remote machines (LP_8 to LP_{11}). When an entity sends an event to another entity, if the two entities belong to the same LP or the two LPs are located on the same processor, the event is inserted into the receiving LP's event list directly. Otherwise, if the channel is classified as a synchronous channel, the event is inserted into EVT_G if the channel spans across different machines, or EVT_L if the channel is between two processors on the same machine. Otherwise, if the channel is an asynchronous channel, and if the channel is between two LPs on two different processors (such as the channel from LP_2 to LP_5), the event is put in the channel mailbox at the receiving LP.

If the channel is connecting to an LP on a remote machine (such as the case from LP_1 to LP_9), the sending LP invokes the `send` function, which puts the event in the remote mailbox The writer thread retrieves the event from the mailbox, serializes the event, and sends it via message passing. To gain better performance, the writer thread may opportunistically pack several events into one message. The reader thread is performing a blocking receive on the remote messages. When a message arrives, which, for example, contains an event from LP_{11} to LP_6, the reader thread deserializes the event and then put the event into the processor mailbox at processor 1. The receiving processor can call the `recv` function to poll the mailbox to see if there are events to be retrieved from the mailbox every time an LP is scheduled to run (at line 11 in Alg. 1). Also, when the processor is running out of ready LPs, it also calls the `recv` function, however, at this time doing a block receive on the mailbox (line 29). When the event is retrieved from the processor mailbox, it is given to the channel mailbox at the receiving LP. Eventually the event is inserted into LP_6's event list.

We use calendar queues [Bro88] for EVT_G and EVT_L, with a bucket size of δ_G and δ_i, respectively. We only distribute the next bucket of future events in the calendar queue at each synchronization point. In this case, the future events beyond the next synchronization window don't need to get sorted and therefore the performance can improve. The all-to-all exchange of the next bucket of events in EVT_L among the processors on the same machine can be done easily via a local barrier though shared memory (line 40). The global all-to-all exchange of the next bucket of events in EVT_G among the machines requires a bit more attention (line 36).

In our implementation, each processor first sends (and counts) the events using the same event delivery mechanism for the asynchronous events (using the same `send` function and going through the writer thread and then the reader thread). Processor 0 then collects the total number of events sent to other machines (using a local barrier) and then performs a global reduce-scatter operation so that each machine ends up knowing exactly how many events it is expected to receive from other machines. The machine waits until all events are received and delivered to the corresponding channel mailboxes before the processors are allowed to continue with the next synchronization window.

3.6.4 Implementation Issues

We consider two implementation issues of the hierarchical composite synchronization algorithm: one on the selection of the thresholds for classifying the communication channels, and the other on the multi-threaded support of MPI on parallel platforms.

Choosing thresholds. The performance of the original composite synchronization algorithm, designed for shared memory, has been modeled analytically; the channel assignment can be formulated as an optimization problem of finding a proper threshold for partitioning the communication channels [NL02]. The hierarchical composite synchronization algorithm extends the original composite approach by introducing an additional threshold for partitioning the cross-memory communication channels. A performance model of the hierarchical algorithm is also available based on linear regression [LR12]. Both models, however, require runtime measurements on the target platform.

Manually performing pilot runs is unattainable in practice when dealing with large-scale models. We notice that the cost structure for local composite synchronization is largely independent from global synchronization. Consequently, we provide an empirical solution to automate the selection of the thresholds by performing measurements separately for local and for global synchronization at the start of simulation. We first set the global synchronization to be asynchronous and then let each machine try out different local thresholds (sampled from the delays of communication channels between LPs on different cores on the same machine). We finally choose the threshold that obtains the best performance. Note that different machines may choose a different local threshold. After that, we fix the optimal local thresholds and then apply the same method to try out different global thresholds (sampled from the delays of communication channels between remote LPs). We finally choose the one that results in the minimum runtime.

Multi-threaded support of MPI. The parallel simulator is a multi-threaded MPI program (using pthreads). *MiniSSF* adopts the SPMD model: each machine runs an MPI instance, which subsequently creates as many pthreads as there are processors and cores on each machine (we call them *work threads*). The logical processes on the machine are automatically assigned to the work threads for parallel execution. In our original design, in addition to the work threads, each MPI instance also creates two additional threads: a *reader thread*, which is responsible for receiving messages from other MPI instances and distributing the received messages to the corresponding logical processes, and a *writer thread*, which is responsible for gathering messages sent from logical processes on this machine to remote logical processes, and sending them to their respective destinations via message passing.

This scheme would work if the particular MPI implementation on the high-end computing platform has full support for multiple threads. Unfortunately, this is not generally the case. We observe that, on most supercomputers we have encountered, the MPI implementations have only limited thread-level support. Some MPI implementations allow only the main thread to call MPI functions, and others, although allowing multiple threads to make MPI calls, permit only one MPI call at a time (that is, the MPI calls must be serialized using explicit thread synchronization).

Requiring full thread-level support would obviously limit the portability of *MiniSSF*. To deal with this problem, we make changes to the original design, and replace the reader and writer threads with one *I/O thread* to handle all MPI calls. The I/O thread applies timed wait on a condition variable (`pthread_cond_timedwait`), which the work threads use to notify the I/O thread that they have messages to send. In addition, we use a facility, called mailbox, to temporarily store the messages sent by the work threads. Upon receiving the notification, the I/O thread will retrieve the messages from the mailbox and immediately send them on behalf of the work threads. When a timeout happens, the I/O thread does a non-blocking check for any messages from other MPI instances (`MPI_Iprobe`). If such a message is found, the I/O thread will retrieve the message from MPI using a blocking receive (`MPI_Recv`) and then deliver the message to the corresponding work thread, using another conditional variable and mailbox.

3.7 Performance Evaluation

To obtain more experience, we investigate the performance of our simulator using three queuing network models. We run our experiments on three distinct supercomputers: Stampede, Kraken, and Blacklight of XSEDE, which is a shared cyberinfrastructure with a collection of high-end computing resources [xse]. The hardware configurations of Stampede, Kraken and Blacklight are listed in table 3.1.

3.7.1 Queuing Models

We create three models. The first model, *the string model*, is used to test the baseline scalability of the simulator. The queues are lined in a circle, each of which is a single-server queue with infinite capacity and an exponentially distributed service time. Upon the departure of a completed job, the queue sends the job either to the previous or the next

Table 3.1: Hardware Configurations of Three Supercomputters

Hardware	**Stampede**	**Blacklight**	**Kraken**
system type	Dell Linux cluster	Cray XT5	SGI UV 1000 Shared-memory
computing nodes	6400	9408	256
processor/core number	two eight-core plus one	two six-core	two eight-core
processor type	Intel Xeon E5 plus Xeon phi	2.6GHz AMD Opteron	Intel Xeon X7560
memory	32GB/node	16GB/node	128GB/node
peak performance	6000 TFlops	1174 TFlops	37.2 TFlops
operating system	Linux CentOS 6.2	Cray Linux	SuSE Linux OS
location	University of Texas	Tennessee Lab and Oak Ridge National	Pitteburgh

queue with an equal probability. In the experiment, we fix a delay of 1 ms between the departure of a job at a queue and its arrival at the next queue. We set the mean service time to be 0.1 ms. At the start, we populate each queue with an average of 10 initial jobs, sampled from a Poisson distribution.

The second model, *the jump model*, has a similar setup. We place an additional link from each queue to another queue randomly chosen within a radius of 100. We also reduce the delay between the queues to be 0.1 ms and increase the mean service time to be 1 ms. We use this model to investigate the performance of our simulator for different computation-communication ratio and communication pattern.

The third model, *the power-law model*, uses a topology with a power-law distribution in the frequency of node degrees, which has been frequently observed in complex networks, such as the Internet. We use BRITE for a hierarchical topology generation [MLMB01]. At the top level, the algorithm generates a random topology consisting of nodes as autonomous systems (ASes) connects based on the Barabási-Albert model. Then, for each AS, it generates a router-level Waxman topology. At both levels, the generator places the nodes randomly in a 2D plane according to a normal distribution and links them with a delay proportional to their Euclidean distance. To represent a connection in AS graph, the generator randomly picks a router within each AS and connects them.

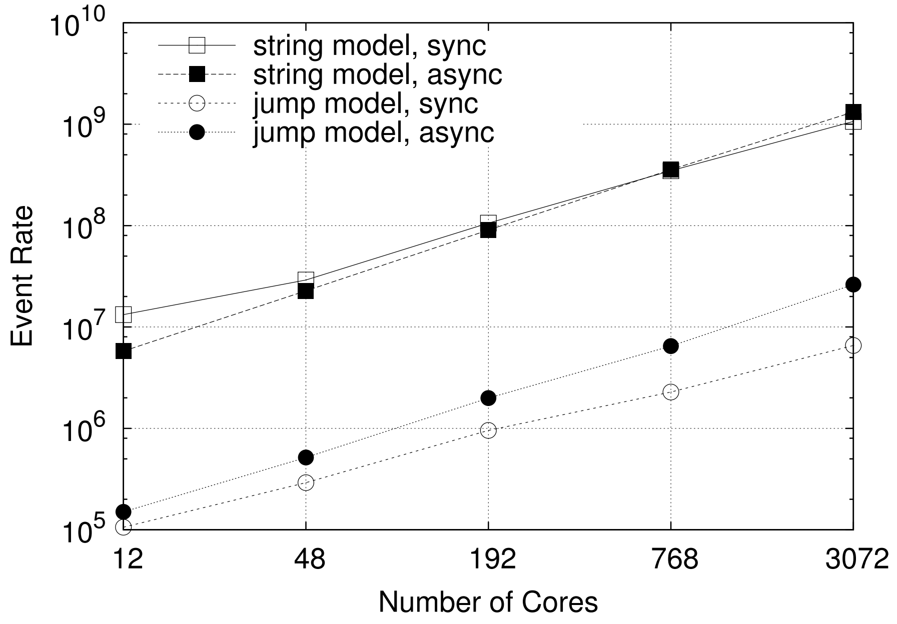


Figure 3.5: String and jump model results on Kraken.

3.7.2 Experiment Results

We first test with the string and jump models on Kraken. We fix the number of queues to be 100 per core and then run the model on 12 to 3,072 cores, doubling each time. For this experiment, we only run *MiniSSF* with the one-level synchronization, either the synchronous (window-based) or the asynchronous (CMB) algorithm, for an initial assessment of the simulator's scaling property. Fig. 3.5 shows the total event processing rate of the simulator as a function of the core count. Note that both axes are in logarithmic scale. The results suggest that the simulator can scale almost linearly on this machine for both models. For the string model, the simulator is able to achieve an event rate of 1.3 billion events per second with the asynchronous method on 3,072 cores. The asynchronous method outperforms the synchronous method with a larger number of cores. For the jump model, the simulator achieves far less event rate due to the model's higher communication to computation ratio. We see that the asynchronous method consistently outperforms the synchronous method, which is due to the model's localized communication pattern.

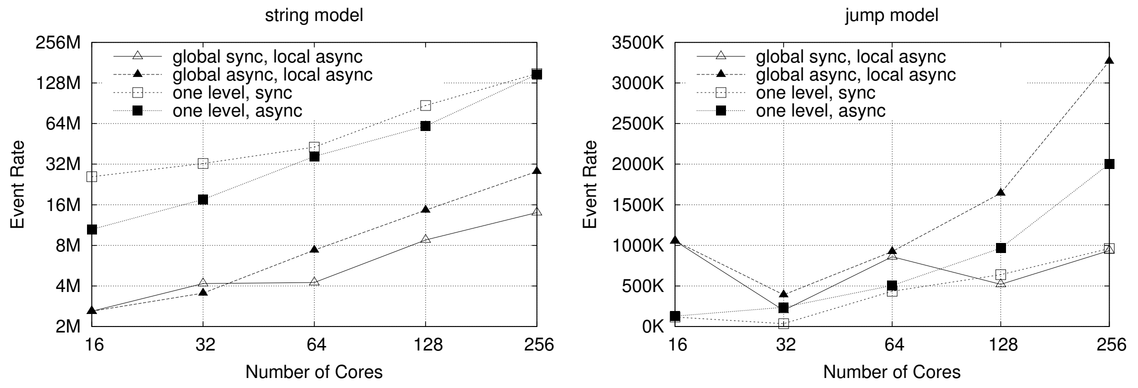


Figure 3.6: Queuing model results on Blacklight.

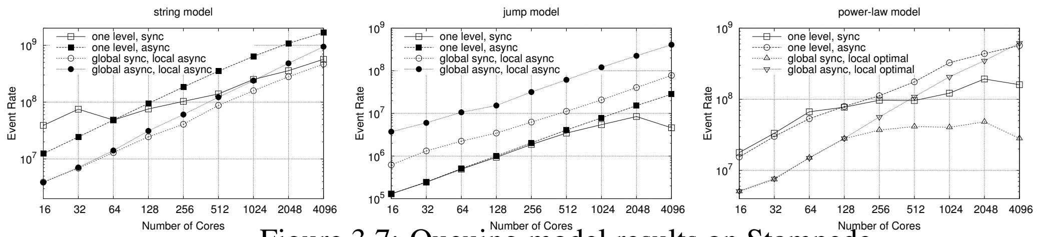


Figure 3.7: Queuing model results on Stampede.

Next, we run the same models on Blacklight and compare the performance between the one-level and two-level synchronization methods. For the two-level synchronization, we configure each machine to run with 16 threads to occupy the available cores. We switch between the synchronous and asynchronous method for the global synchronization and use the asynchronous method at the local level as it produces better performance. We increase the number of cores from 16 to 256. Fig. 3.6 shows the results. The one-level methods exhibit better performance than the hierarchical approach for the string model (the left plot), achieving as much as 150 million events per second. But the situation is reversed from the jump model (the center plot) where the global asynchronous and local asynchronous method significant outperforms the rest. Speculating from this quite opposite outcomes, the benefit from the hierarchical synchronization seems to be less obvious for compute-intensive models.

The left and center plots in Fig. 3.7 show the results from running the string and jump models on Stampede, respectively, as we simultaneously increase the model size and the number of cores (from 16 to 4,096 cores). Except for the one-level synchronous method, the performance scales up almost linearly. The maximum event processing rate is 1.7 billion events per second on 4,096 cores.

At last, we run the power-law model on Stampede. For the experiment, we fix the number of routers within each AS to be 1,000. We set the number of ASes to be the same as the core count. We set the parameters of the topology generator so that the delays for the inter- and intra-AS links are both normally distributed with a cutoff range from 0 to 12 ms. We set the mean service time at each queue to be 0.1 ms. The right plot in Fig. 3.7 shows the results of running the simulator both with one level, either synchronously or asynchronously, and with two levels, where the global level is either synchronous or asynchronous, and the local level uses a threshold empirically determined to obtain the optimal performance. The performance of the two-level method with global asynchronous and local optimal threshold increases faster with the core count and eventually achieves the best performance on 4,096 cores.

From these experiments with different scenarios setups, we get the basic idea of how to configure the simulator to achieve the performance as good as possible, for different models running on various platforms.

3.8 Conclusion

This chapter presents a hierarchical composite synchronization method, which classifies the channels between the logical processes as either synchronous or asynchronous according to the channel latencies. The composite method uses window-based synchronization to handle the delivery of events across the synchronous channels, and uses null-message-based synchronization to ensure the delivery of events across the asynchronous channels. In doing so, the composite approach can potentially avoid the cost of performing frequent collective operations for the traditional window-based method, where the synchronization window is determined by a few small channel delays; and it can also reduce the channel scanning cost for the traditional null-message-based method when dealing with dense model graphs, where some of the channels with large channel delays can be handled more efficiently using collective operations. The hierarchical approach ensures that the algorithm makes the important distinction between synchronizing logical processes located on different distributed-memory machines using expensive message passing mechanisms, and those located on different processors or cores at the same machine via efficient shared-memory-based inter-process communication. Consequently, our algorithm provides separate mechanisms for handling global synchronization for distributed-memory machines and local synchronization for shared-memory multiprocessors and multicores.

We also present a simple performance model for the hierarchical composite algorithm. For practical purposes, we describe a method of using linear regression to find the parameters of the performance model for a target parallel platform. We make attempts to use the analytical model to predict the performance of the simulation. Although the prediction value has a certain difference from the real value, it correctly track the curve of real performance. One can use the model to derive the best configuration of the simulation. The preliminary experiments also show that our new algortithm under predicted configuration achieve significantly better performance than the pure synchronous and asynchronous approaches used separately for global and local synchronization.

At last, we describe the parallel discrete event simulation core that includes hierarchical composite synchronization. We believe *MiniSSF*'s simplistic and yet powerful API will provide a general appeal to the simulation practitioners, who will find it easy to develop models that can be run both on common desktops and on modern supercomputers. The simulator is open-source and can be freely obtained at `http://www.primessf.net/minissf`. To advocate process-oriented simulation for its expressive power, the simulator has a fully automated compiler-based source-code translation scheme supporting efficient user-space multi-threading. Equally important is the simulator's ability to automatically adapt its synchronization to reflect the model's computation and communication demands as well as performance characteristics of the underlying parallel platform.

Algorithm 1 The hierarchical composite synchronization algorithm running on machine *i* and processor (or core) *j*

```
1: $Q_{ready} \leftarrow \phi$; $EVT_G \leftarrow \phi$; $EVT_L \leftarrow \phi$
2: for all LP $v_x \in V_{ij}$ do
3:     $t_x \leftarrow 0$; $t_{e(y,x)} \leftarrow l_{e(y,x)} \forall e(y,x) \in I_x$
4: $t \leftarrow 0$; $w_G \leftarrow \delta_G$; $w_L \leftarrow \delta_i$; $w \leftarrow \min\{w_G, w_L\}$
5: while ($t < T_{term}$) do
6:     for all $v_x \in V_{ij}$ do insert LP $v_x$ into $Q_{ready}$
7:     $done \leftarrow 0$
8:     while ($done < |V_{ij}|$) do
9:         while ($Q_{ready}$ is not empty) do
10:            remove LP $v_x$ with smallest $t_x$ from $Q_{ready}$
11:            $LBTS = \min_{e \in I_x}\{t_e, w\}$
12:            process events in LP $v_x$'s event list until $LBTS$
13:            $t_x \leftarrow LBTS$
14:            for all $e(x,y) \in O_x$ do
15:                $t_{e(x,y)} \leftarrow t_x + l_{e(x,y)}$
16:                if ($v_y \in V_{ij}$) then
17:                    $LBTS = \min_{e \in I_y}\{t_e, w\}$
18:                    if ($t_y < LBTS$ and $v_y \notin Q_{ready}$) then
19:                        insert LP $v_y$ into $Q_{ready}$
20:                else
21:                    send null message updating $t_{e(x,y)}$
22:            if ($t_x = w$) then $done \leftarrow done + 1$
23:        if ($done < |V_{ij}|$) then
24:            wait for incoming null message updating $t_{e(y,x)}$
25:            if ($t_x < \min_{e \in I_x}\{t_e, w\}$) then insert LP $v_x$ into $Q_{ready}$
26:    $t \leftarrow w$
27:    if ($w = w_G$) then
28:        $w_G \leftarrow w_G + \delta_G$
29:        all-to-all exchange of evts in $EVT_G$ among all machines
30:    if ($w = w_L$) then
31:        $w_L \leftarrow w_L + \delta_i$
32:        all-to-all exchange of evts in $EVT_L$ among local processors
33:    $w \leftarrow \min\{w_G, w_L\}$
```

CHAPTER 4

DISTRIBUTED AT-SCALE EMULATION WITH SIMULATION SYMBIOSIS

In this chapter, we apply the existing symbiotic approach [ERL15] to improve a specific network emulator—Mininet. With effectively integrating emulation with simulation, we can improve the scalability of models, validate the design and implementation, and conduct distributed emulation on any cyberinfrastructure testbeds. We called the hybrid experimental system *mininet-symbiosis*. With this system, one can use Mininet to run applications directly on the virtual machines and software switches, with network connectivity represented by detailed simulation at scale. We also propose a method for using the symbiotic approach to coordinate separate Mininet instances, each representing a different set of the overlapping network flows. In this case, one can more effectively study the behavior of real implementation of network applications on large-scale networks, since the interaction between the Mininet instances is only capturing the effect of contentions among network flows in shared queues, as opposed to having to exchange individual network packets, which can be limited by bandwidth or sensitive to latency. We provide a prototype implementation of the new approach and present validation studies to show it can achieve accurate results. Additionally, We present a case study that successfully replicates the behavior of a denial-of-service (DoS) attack protocol.

4.1 Introduction

During the last ten years, significant advances have been made in Future Internet Architecture (FIA) design and cyber-infrastructure development. Large-scale coordinated efforts (such as [Mob, Nam, eXp, NEB]) with bold ideas, innovative and oftentimes disruptive designs have been proposed, in order to provide secure, high-performance and ubiquitous services for applications of the fututre. For example, the Named Data Networking (NDN) proposed to decouple the trust of data from the trust in hosts and servers by naming the data instead of their location traditionally. Several radically scalable communication mechanisms such as content caching, multipath routing can be carried out correspondingly. Therefore, all these ideas express the design principles of the Future Internet—globalized, flexibility, adaptability, which will transform our lives further.

Essential to the FIA research is the development of network testbeds that can validate key design decisions and expose operational issues at scale. As introduced in 2, federate cyber infrastructure such as GENI, NSFCloud [GENb, Clo, Cha] has made signficant development that are broadly used by researchers to conduct networking experiments. While all these efforts would pave the way for the network researchers (as well as the network engineers) to validate design and implementation issues directly on the cyber-infrastructure testbeds, one needs to understand the deficiencies of solely relying on real-world implementation and physical deployment in network studies. We illustrate this important issue through a few hypothetical examples:

- A new robust map-reduce algorithm [DG04] needs to be evaluated for multi-tenant cloud computing environments. The performance of the algorithm depends on the job characteristics (such as the distribution on the number of jobs and the individual job sizes), as well as the configuration and stability of the available resources of the cloud platform. One would find it extremely time-consuming to explore the entire algorithmic parameter space on physical testbeds; let alone the highly diverging cloud configurations.
- An enterprise network traffic engineering solution based on OpenFlow [MAB^{+}08], which uses opportunistic traffic load balancing and multi-path schemes to increase the throughput of heavy-hitter flows, has been proposed. Important questions remain unanswered—for example, whether this algorithm is robust under various traffic conditions, whether the algorithm would perform well due to partial deploy-

ment with varying proportions of non-cooperative entities, and whether the algorithm could scale out to a larger number of ISPs.

- A data center transport-layer protocol has been proposed (similar to [MBI$^+$14]), which is expected to both reduce flow completion time and increase data throughput. The algorithm has been implemented and tested in a small-scale homespun DCN testbed; one needs to know whether it is ready for deployment in a production data center. Before that, however, one would like to investigate the algorithm's optimal performance conditions for the large data center with high bisection network capacity and also with various traffic loads with known stochastic properties.

These examples highlight some of the intrinsic limitations of cyber-infrastructure testbeds. No matter how useful are they, they still have the inherent deficiencies as physical testbeds. They are limited in scale; it is thus difficult, if at all possible, to reveal scaling properties and robustness issues. They also lack flexibility: it is cumbersome and time-consuming to set up experiments to explore the design and configuration space given the large set of control parameters and system configurations. One would also find it difficult to test algorithms and applications beyond the existing setup of the physical environment. This would in turn limit the researcher's ability to investigate network applications under alternative conditions and ask what-if questions.

To overcome these problems, network researchers used to conduct evaluative studies by using simulation and physical testbeds together. Simulation is usually used to validate the key functions under various network scenairos, or run large-scale models. Physical testbeds are used for small-scale real-world studies. There are two problems associated with this complementary approach. First, the researcher typically use simplified models in simulation, which can not reveal the realistic behavior of the target applications or protocols at scale. Second, network effect from coexisting applications can definitely not be reproduced in simulation, which is the necessary context for networking evaluations.

For example, an enterprise network traffic engineering solution can be heavily dependent upon the behaviors of the users and the characteristics of the prevailing applications.

Another popular method is to use emulation. For example, Mininet [LHM10, HHJ+12] is a popular emulator that can prototype networks on a laptop by using a lightweight container-based virtualization. We have described in chapter 2 it is relatively easy to build and execute experiments with Mininet. However, it is well-known that Mininet only provides a limited capacity for both CPU and network I/O. Consequently, it does not work well on large scenarios and topologies with large volume of traffic, even if used in a cluster environment.

The above problems call for a method to organically integrate physical testbeds and simulation/modeling for network experimentation. Previously we proposed a symbiotic approach to combine both simulation and emulation [ERL15], which each can benefit from the other. Both systems evolve in real time. The simulation system benefits from the emulation system by considering real network traffic generated by the unmodified software directly executed on real systems. The emulation system benefits from the simulation system by receiving network updates and using it to calibrate communication between the real applications. As a result, the symbiotic approach allows us to test and analyze applications by *embedding* them seamlessly in diverse virtual network settings. The symbiotic approach is the foundation of this work, we will present it specifically in section 4.2.

For this work, we apply the symbiotic approach to combine Mininet with simulation [LMAR15]. By using this hybrid approach, one can use Mininet to run applications directly using the virtual machines and software switches. These virtual machines can be a part of a large-scale network simulated by the network simulator for representation of diverse network scenarios. This would allow us to efficiently and accurately incorporate complex network models, such as different network topologies, network-wide traffic matrices, as well as stochastic models to describe user demands, mobility, and applications behaviors. The essential aspect of our approach is to migrate redundant traffic from emulation to simulation. Rather than re-creating each network packet generated from applications, we can capture in real time the aggregate traffic demand of these applications and simulate the corresponding effect on the network queues (effective bandwidths, packet loss, and packet delays), that can affect other applications. We also propose a method for using the symbiotic approach to coordinate separate Mininet instances, each representing a different set of the overlapping network flows. By effectively distributing network emulation among separate machines, one can significantly improve the scalability of the network experiments.

The specific contributions of this work are two aspects. First is the design and implementation of the symbiotic construct that can effectively integrate the emulation testbed with a network simulator so that one can test applications and algorithms realistically with various system configurations and design parameters. Second is the coordination of separate Mininet instances that represent a different set of real flows so that one can conduct hybrid at-scale experiments distributedly with any cyber infrastructure. Our system also has the potential to study innovative SDN/OpenFlow applications in the future.

4.2 The Symbiotic Approach

The approach [ERL15] previously proposed to form a symbiotic relationship between network emulation and simulation, aims to conduct high-fidelity high-performance network experiments. The methodology enlights this work, so we present its main idea here.

The system prototype consists of two parts: a simulation system and an emulation system. We use *the simulation system* to run the full-scale network model in real time with detailed network topology and protocols for a close representation of a target network. We use *the emulation system* to inspect the detailed behavior of the real applications, where a number of nodes in the target network can be selected as "emulated" nodes to run unmodified software directly on the virtual machines with specified operating systems, real network stacks, libraries and software tools. The full-scale network is simulating as "virtual network"; the components except for emulated ones are "simulated hosts, routers, or traffic". The emulated hosts and routers will be instantiated on either physical or virtual machines in the emulation system, which will run "target applications" such as web clients/servers, peer-to-peer applications, routing algorithms and so on. They are still need to be represented in simulation. Since the traffic of target applications may mix with the simulated traffic in virtual network, one must be able to accurately capture the effect of the simulated flows on the emulated flows, and vice versa.

In the symbiotic approach, the simulation and the emulation forms a closed-loop of communication to represent the true state of the target applications in the virtual network. The communication includes two aspects: the two systems must accurately exchange their state; they also need to synchronize efficiently. Otherwise, the realism and the scalability of the experiments will both be impaired. The work [ERL15] proposed several approaches regarding to the synchronization between two systems; we succinctly introduce them here.

- To reduce the complexity of emulation system, the model instantiated in the emulation is a downscale topology. The reduced model only contains the network links that traversed by emulated flows. Then the set of links traversed by the same emulated flows are compressed into one network segment. One can reduce the network path between emulated hosts to contain less segments. Finally, the emulated hosts and routers will be instantiated on individual machines. The network segments will be represented as "pipes" on a delay node, which has the software constructs to modulate the pass by traffic.

- The primary purpose of this approach is to make the real network traffic between the emulated hosts and routers probabilistically experiencing the same delays and losses as if the target applications were directly connected by a real full-scale network. A queuing model was proposed to accurately apply the virtual network states, including packet drop probability, packet delay into the downscale model. As in the downscale model, emulated hosts and routers are connected by pipes. The emulated packets flowing through these pipes can be dropped or added with artificial delays to reflect the simulated network conditions. The state of the pipes will be updated constantly in real time by applying the queuing model on statistics collected from the simulated counterpart. We elaborate this model in next section 4.3.
- To efficiently represent application traffic in simulation, the work [ERL15] proposed to report the traffic demand periodically from emulation to simulation instead of injecting every packet.

A prototype is built with dummynet [Riz97] and PRIME [Liu08] to realize the symbiotic idea. Some tests have been conducted for validation purpose in [ERL15]. The work [CR10] a dummynet node has the capability to handle the traffic of 2-300Kpps. Therefore, the prototype designs all the emulated hosts running target applications with a centralized dummynet node, which will not be able to scale up well of applications for globalized networks.

4.3 Mininet Symbiosis

In this section, we discuss our design for improving the capability of an emerging emulator, *Mininet*. We call this new hybrid system as *mininet symbiosis*.

4.3.1 System Overview

Mininet is a popular container-based emulator for testing OpenFlow applications. It uses lightweight OS-level virtualization to emulate the hosts. Each virtual host corresponds to a container attached to a separate network namespace (a mechanism introduced since Linux kernel 2.6.24). Each network namespace can contain a virtual network interface with a distinct IP address along with independent functions of the TCP/IP stack (such as the kernel routing/forwarding table). The virtual network interfaces can be connected via virtual Ethernet links to the software switches (i.e., OVS instances), augmented with OpenFlow capabilities. An OpenFlow controller can be connected to the OpenFlow-enabled software switches for a full implementation of the software-defined networking experiment. A significant portion of the Mininet implementation is a python library to assist the users to create and maintain the virtual network topology for emulation. Mininet uses `cgroups` for scheduling and resource management so that one can limit the CPU usage for all processes belonging to each container. Mininet also uses `tc`, the Linux traffic control, to control the link properties, such as link bandwidth, packet delay, and packet loss.

A typical procedure for using the symbiotic approach can be shown more easily through an example. Our goal is to execute the target network applications (`iperf` for a simple example) in Mininet containers while creating an illusion that these applications are running on an arbitrary network. Our approach starts by first having the user to specify a network model, which includes a simulated network topology (on which the target real applications are expected to run), as well as network protocols and applications, and how they are engaged during the experiment. For example, one can incorporate complex network topologies with stochastic models for network-wide traffic generation. Fig. 4.1 shows a simple virtual network with four routers connecting many hosts.

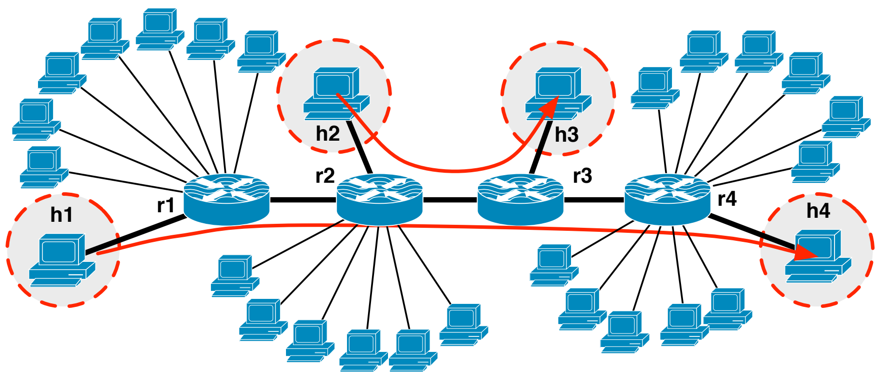


Figure 4.1: A target virtual network with emulated traffic identified.

Next, the user can identify a subset of hosts to be emulated in Mininet (we call them *emulated hosts*). They will be instantiated as containers and therefore capable of directly running the target network applications. To reduce overhead, we also ask the user to identify flows that will be generated between the emulated hosts during the experiment (we call *emulated flows*). This can significantly reduce the facilities that need to be maintained for symbiosis. In the example shown in Fig. 4.1, we specify two emulated flows: one from h1 to h4, and the other from h2 to h3. Here again, for brevity, we only show one-directional traffic. Most flows (such as TCP) would be bi-directional, in which case the user would need to specify the flows for both directions.

Afterwards, we invoke a process, called *downscaling*, in which the original full-scale network simulation model together with the identified emulation traffic is processed to produce an reduced emulation model for Mininet. As have already described the downscaling method, we directly jump into the finalized model for our example in emulation. The downscaled emulation model (only forwarding portion) of the same example is shown in Fig. 4.2, which consists of the four emulated hosts and two switches, connected by five network pipes.

In [ERL15], we derived a closed-form solution, by which the traffic conditions such as packet delays and losses in the full-scale simulated network can be summarized to control

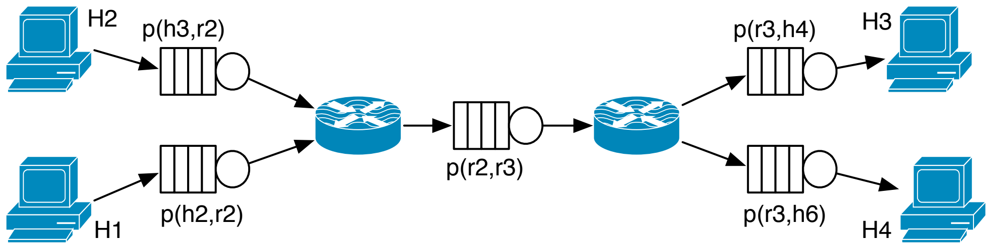


Figure 4.2: A downscaled network model to run in Mininet.

the real traffic in the downscale emulated network. We capture the main results below and explain how to apply it in our *mininet symbiosis*.

In general, let $q_1, q_2, \cdots, q_n$ be the list of network queues in simulation that are supposed to be traversed by the real network traffic. In simulation, we collect three measurements for each queue q_i and periodically report them to the emulator:

1. We measure p_i, which is the average drop probability due to buffer overflow;
2. We measure λ_i, which is the arrival rate of the regenerated emulated network flow; and
3. We measure w_i, the average packet queuing delay.

Once these measurements are propagated to the emulator, we can calculate the packet drop probability for the network pipe:

$$p = 1 - \prod_{i=1}^{n} (1 - p_i) \tag{4.1}$$

And we can calculate the service rate (i.e., the bandwidth) of the network pipe:

$$\mu = \frac{\lambda_p (\Delta T + W_2 - W_1)}{\Delta T \left(1 + W_1 \lambda_p - \sqrt{1 + W_1^2 \lambda_p^2}\right)} \tag{4.2}$$

where $\lambda_p = \min_{1 \le i \le n}\{(1 - p_i)\lambda_i\}$, which is the minimum effective arrival rate at all queues; ΔT is the sample interval (say, 100ms), which is also the interval at which the simulator updates the emulator with the measurements; $W_1 = \sum_{1\leq i\leq n} w_i$ is the total queuing delay through the n queues measured in simulation; and W_2 is the average packet queuing delay through the corresponding network pipe measured in emulation.

By design, our symbiotic system *mininet-symbiosis* consists of a simulation system and an emulation system running side by side. The simulation system is a real-time network simulator (we use PrimoGENI [VEL11] for our prototype implementation), and the emulation system consists of one or more Mininet instances, potentially running on separate machines (see Fig. 4.3). Communication between the real-time network simulator and the Mininet instances is achieved via TCP connections, whereas the simulator functions as the server and each Mininet instance as a client. The real-time network simulator runs the original full-scale network; as such, it needs to implement necessary network elements (such as routers, hosts, network interfaces and links) and common network protocols (such as IP, TCP, UDP, and others). In addition, two components are added to the simulator to facilitate synchronization with the Mininet instances: a traffic monitor and a traffic generator. The traffic monitor is used to collect measurements at each queue q_i traversed by the emulated flows, which include the packet drop probability p_i, the arrival rate of emulated flows λ_i, and the queuing delay w_i. These measurements are collected periodically every ΔT units of time and then sent to the corresponding Mininet instances. The traffic generator receives information from Mininet about the traffic demand d_k from applications for each emulated flow k in terms of the number of bytes requested to be sent during the last interval. Upon receiving this information, the simulator generates the emulated flows by initiating the corresponding TCP or UDP sessions in simulation with the same demand size accordingly.

In Mininet, the emulated hosts are instantiated as Linux containers with separate network namespaces, and the switches are represented by OVS instances. The virtual Ethernet (`veth`) pairs are used to represent the links augmented with the Linux traffic control

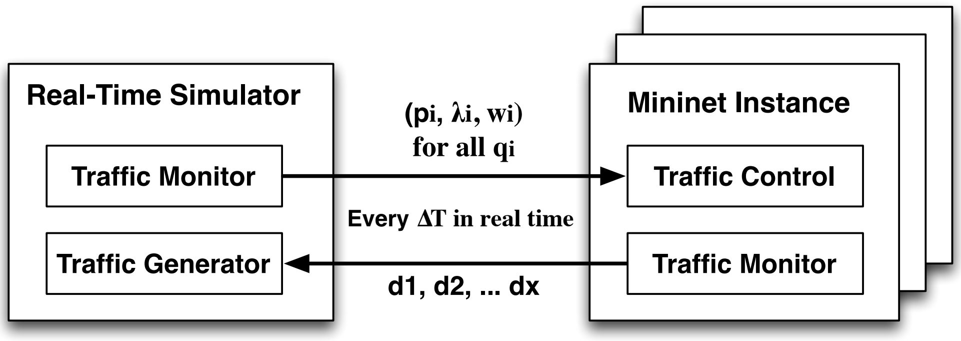


Figure 4.3: Mininet symbiosis setup.

(`tc`) for managing the link properties. Linux `tc` is a set of tools (included since kernel 2.2) to allow users to have fine-grained control over the packet transmission. Linux `tc` consists of different queuing mechanisms, easily composable for handling more complex situations (including packet mangling, IP firewalling, and bandwidth metering). We use `tc` for setting the link bandwidth, the packet delay, and the random packet loss probability. More specifically, we statically set the link delay as the cumulative propagation delay of the links between the consecutive queues that constitute the network pipe. We modify the packet loss probability and the link bandwidth dynamically during the experiment using the measurements from simulation (Equations 4.1 and 4.2).

Note that our symbiotic approach can easily support distributed emulation, where multiple Mininet instances can operate in parallel, each handling a different set of emulated flows. For the example shown in Fig. 4.1, the flow from h2 to h3 can be emulated in a separate Mininet instance from the one used for emulating the flow from h1 to h4. The downscaled models for the two Mininet instances are shown in Fig. 4.4. Note that the state of the network pipe, $p(r2, r3)$, is mirrored on both instances; that is, they will be controlled by the simulator with the identical link properties.

In the following sections, we discuss the detailed design and implementation of the symbiotic constructs.

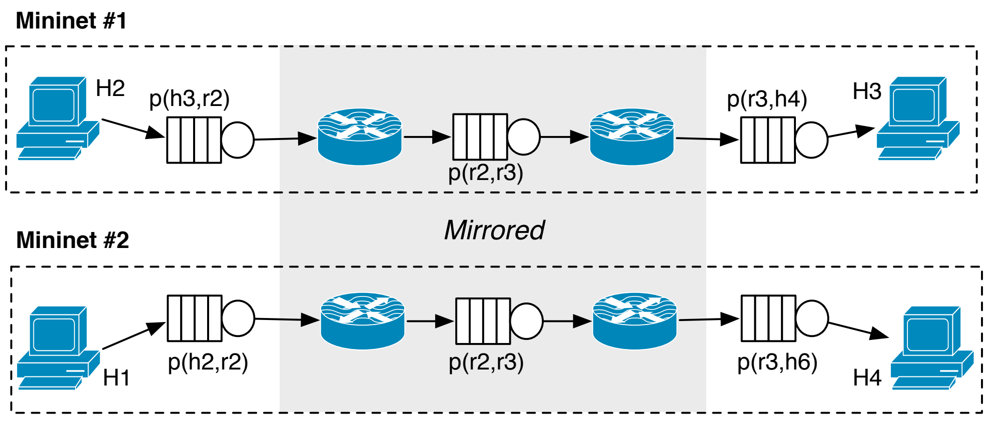


Figure 4.4: Downscaled models for two Mininet instances.

4.3.2 Regenerate Emulated Flows in Simulation

A unique aspect of our symbiotic approach, different from the traditional real-time network simulation method, is that real network packets in the emulated system that need to be simulated on the full-scale network do not need to be captured individually to reproduce the same traffic effect (in order to calculate their packet delays and packet losses accordingly). Instead, the symbiotic approach reproduces the effect of the real traffic flows in simulation by having the emulation system to capture the interval-based traffic demand at the traffic source (preferably at the application/transport interface) and then reproduce the demand traffic using the corresponding simulated TCP or UDP. In this case, we can minimize the synchronization overhead between the simulator and the physical system.

There are several ways to collect traffic demand agnostic of specific application behaviors. A somewhat complicated method involves creating a wrapper to a socket library and collect the read/write and send/receive calls from the applications right before they invoke the kernel functions. Another possibility is to monitor the state of a TCP connection using the `tcpprobe` kernel module. One can monitor the `SND.NXT` pointer, which represents the sequence number of the first unsent byte of user data and then calculates the difference between consecutive packets to estimate the demand over an interval. The drawback of this approach, however, is that this demand (taken from consecutive packet departures) represents a transmission that has already taken place from the perspective of the transport layer. With zero lookahead for reproducing the traffic, the system would be sensitive to the latency between the simulator and the Mininet instances.

Our traffic monitor on Mininet uses a simple and lightweight solution to capture the traffic demand at each Linux container (emulated host). We chose to use a tracing tool for the Linux system calls, called `strace`. One can use `strace` to collect the traffic demand at the interface between the applications and the transport layer. Network system calls—such as `connect`, `accept`, `read` and `write`, and others—invoked by applications running inside the containers can be captured and parsed continuously to arrive at the application traffic behavior. The following shows a snippet of the `strace` output for running `iperf` data transfer inside a container. We can see that the `connect` system call from process 15742 (which is the `iperf` process) established a TCP connection with another container with the IP address 10.0.0.2. The subsequent system calls to `write` indicate the request to send 131,072 bytes of data each time via the TCP connection.

```
[pid 15742] connect(3, {sa_family=AF_INET,
    sin_port=htons(5001), sin_addr=
    inet_addr ("10.0.0.2")}, 16) = 0
[pid 15742] write(3, ... 131072 <unfinished ...>
[pid 15742] <... write resumed> ) = 131072
[pid 15742] write(3, ... 131072) = 131072
[pid 15742] write(3, ... 131072 <unfinished '...>
```

All socket-related system calls can be captured in this way. For certain system calls, such as `write` , we need to distinguish the calls handling data transmissions over sockets from those handling regular file IOs. This can be achieved by checking the state of the file descriptor of a process in the Linux /proc system. For example, the information for the `iperf` process can be located at `/proc/15742/fd/3`. Each container in Mininet starts with a bash shell. To speed up the process, we can cache the lookups for child processes spawned from the container's bash process, so that one can quickly identify the connections used by the applications running by a child. The use of `strace` is indeed lightweight. In our prototype, we found that the overhead is only around 1% CPU per container.

Once the demands are received, in order to generate the same amount of simulated traffic, we instantiate a "symbiosis application" at each of the emulated hosts at the start of the simulation. For each emulated flow, the symbiosis application at the sender host creates a socket connection (either a TCP or UDP session) with the receiver also at the start of the simulation. During the experiment, upon receiving an updated traffic demand from the emulator, the simulator simply issues a send command with the same size for the corresponding session at the sender host. Note that in order to preserve the same traffic behavior, the real-time network simulator must support the same set of TCP variants commonly used in the physical platform. PrimoGENI contains fourteen TCP variants that can be found commonly in use today, including New Reno, BIC, CUBIC, and others. These TCP congestion control mechanisms have been previously ported from the Linux implementation and have been tested extensively [ELL09].

4.3.3 Actuate Network Pipes

As mentioned earlier, the simulator is instrumented to generate the queuing statistics at the simulated network interfaces that constitute the network pipes, including the packet loss probability, the packet arrival rate, and the average queuing delay. These measurements are distributed periodically to the corresponding Mininet instances that handle the network pipes.

The network pipes are created with Linux `tc` using the specific token bucket queuing disciplines. The delay of a network pipe is fixed at the system configuration; the value is the total propagation delay of the network links that constitute the network pipe in the simulation model. The packet drop probability and service rate need to be changed during the experiment. It is important that, once the simulation measurements reach the Mininet instances periodically (say every 100ms), it is necessary to change the corresponding `tc` link properties immediately so that the real traffic flows can reflect the traffic conditions in the simulated network. In Mininet, we created a separate thread to receive the periodic updates from the simulator: (p_i, λ_i and w_i), for each simulated queue q_i traversed by the emulated flows.

The packet drop probability can be applied directly using the `replace` primitive in `tc`. Using `tc replace` is fast and convenient. To verify its effectiveness, we tested by executing the `tc show` command immediately after applying `replace` primitive. We did not notice any degradation in traffic performance for all experiments we performed even with update small update intervals.

In order to apply Equation (4.2) to calculate the new service rate, we need to measure the average packet queuing delay, W_2, through the network pipe. Directly measuring the packet queuing delay by packet can be costly. Instead, we can estimate the average queuing delay by sampling the instantaneous queue length gathered from the `tc` statistics. We created a collector mechanism that obtains the relevant values from the kernel. Since these values are constantly monitored for the Linux queues in any case, the collector presents no additional overhead. In particular, we capture instantaneous queue lengths in bytes at smaller sample intervals (say, one tenth of the update interval used to synchronize simulation and emulation). We accumulate the samples and average them over the update period. The resulted average queue size is then divided by the service rate to produce the estimated average queuing delay W_2. Finally, we can apply Equation (4.2) to calculate the new service rate. Again, we use `tc replace` to update the network pipe.

4.4 Experiments for System Validation

In this section, we first conduct experiments to validate our design using a prototype implementation. We aim to investigate whether our system is robust and can accurately capture the interaction between simulated and emulated traffic in the reduced model as in the full-scale model. For that, we first investigate the correctness for individual interaction between two subsystems—regenerating the emulated traffic in simulation and reflecting the network effect in emulation. Then we use a simple model to validate the entire system.

4.4.1 Reproducing Emulated Traffic

We first study the effectiveness of the mechanisms for reproducing the emulated traffic demand in the simulation. In particular, we aim to examine whether the real traffic demand from the virtual machines can be captured accurately by our traffic monitor in Mininet, and whether the new traffic generator module in the simulator can faithfully reproduce the same flows in a timely fashion. We start with experiments of a single pair end hosts running real applications on one Mininet instance and extend to multiple pairs of source and destination of several Mininet instances running on different virtual machines.

Single Emulated Traffic

We used a simple dumbbell model, similar to the one shown in Fig. 4.2. We set the bandwidth of the "bottleneck" link connecting the two routers to be 10 Mbps, and all other "spoke" links to be 1 Gbps. The bottleneck link has a propagation delay of 15 ms

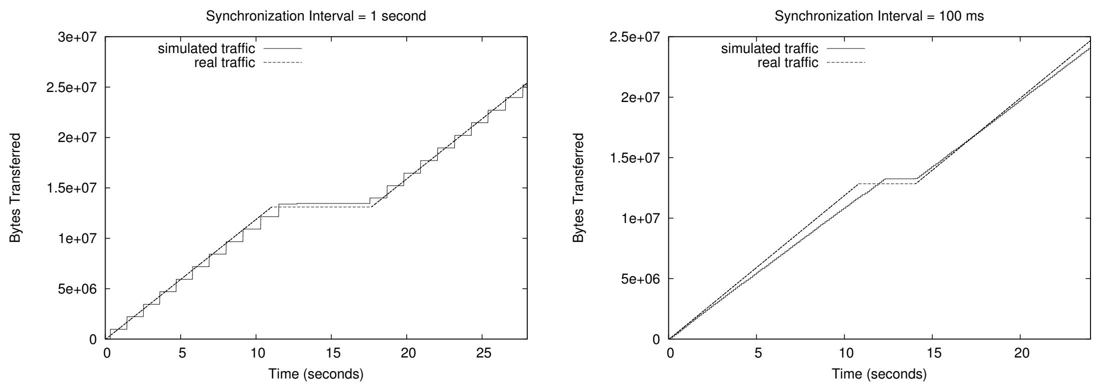


Figure 4.5: Reproducing real traffic in simulation.

while the spoke links all have a propagation delay of 1 ms. We ran the real-time simulator and the Mininet instance on separate machines connected via a gigabit network.

In the first experiment, we manually created two TCP flows using `iperf` one after another with only a few seconds in-between. The two flows were generated from the same emulated host on one side of the dumbbell to a fixed host on the opposite side (thus traversing the bottleneck link). We ran `tcpdump` to capture the packets at both the sender and the receiver, and therefore used the TCP sequence numbers to measure the traffic situation in Mininet. We compare them against the corresponding traffic regenerated in simulation.

We started by using one second as the interval for synchronizing the simulator and the emulator; it's at least one order of magnitude higher than the network latencies one would normally observe over the wide-area network. The result is shown in the left plot of Fig. 4.5. The staircase behavior of the simulated traffic is due to the large synchronization interval. The traffic demand from Mininet is only reported to the simulator once every second. As a result, the simulator tried to replay the entire one second worth of traffic at the beginning of each interval. Despite this artifact, however, the simulated traffic is shown to be able to track the real traffic quite well.

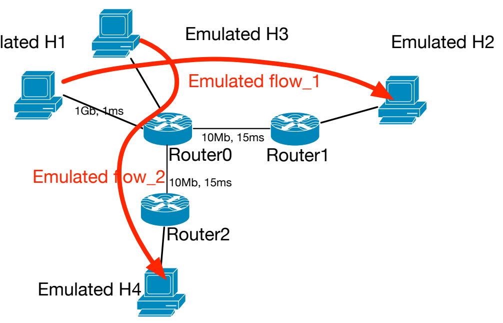


Figure 4.6: Real Traffic from two mininet instances model.

Next we reduced the synchronization interval from one second to 100 ms and performed the same experiment. The result is shown in the right plot of Fig. 4.5. The previous staircase behavior of the simulated traffic is no longer apparent. We observe that the simulated traffic can still match with the real traffic, however with a slight decrease in its transfer rate. This is due to an issue with the simulator's traffic generator. In the original design, we extended a simple server-client model in the simulator, where a request message has to be sent from the client to the server, which would cause a slight delay before the data transfer can be effectuated. There is also additional overhead related to the choice of using a smaller segment size for TCP. We are redesigning the simulation traffic generator to remove these problems.

Multiple Emulated Traffic

To minimize side effect between different traffic, we used another model that has non-overlapping paths for each set of flows in this experiment. The model is shown in Fig. 4.6. We set the bandwidth of the links connecting the two routers to be 10 Mbps, and all other links to be 1 Gbps. The links between two routers have a propagation delay of 15ms while others all have a propagation delay of 1ms. Two set of target application traffic`iperf` are going between emulated hosts h1 and h2, h3 and h4. We used two Mininet instances on different virtual machines (or physical machines) to instantiate separate reduced models

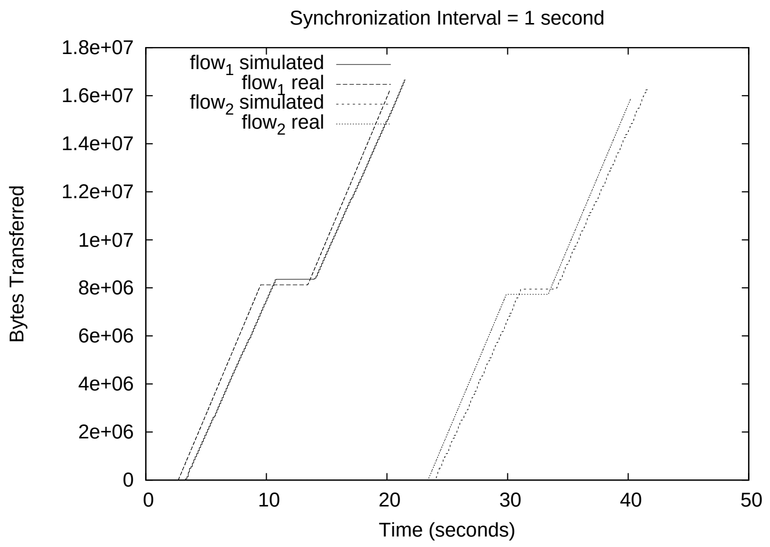


Figure 4.7: Reproducing real traffic from two Mininet instances in simulator.

of h1, h2 and h3, h4. We first created two TCP flows start from h1 and destinate to h2, with a few seconds in between; then two flows from h3 to h4. The two batches of traffic neither share the same bottleneck, nor have overlapping running time. We wanted to isolate them, in order to make a simple scenario.

In this case, we are still using TCP sequence number from Mininet side to compare with traffic measurement from simulator side. For achieving better performance, we choose 100 millisecond as the synchronization interval. The result is shown in Fig. 4.7. From the results, we can see both sets of traffic can be triggered in the simulation and can be generated accordingly.

4.4.2 Representing Simulated Network Effect

Second, we studied the capability of reflecting the influence from all other traffic to target application traffic. Our queuing model is designed to capture this influence expressed as packet drop probability and service rate. Then emulated Traffic is regulated accordingly. We are conducting experiments to investigate two parts: the effectiveness of our

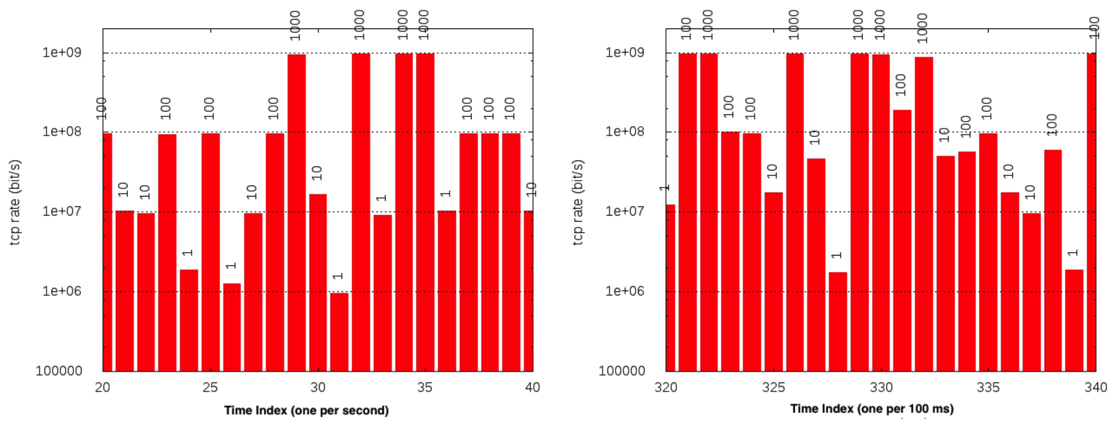


Figure 4.8: Controlling traffic in Mininet.

traffic control in Mininet, the robustness of capturing traffic fluctuation from simulation to Mininet.

Traffic Control Validation

In this experiment, we started a long-term TCP flow between two virtual machines using `iperf`. The two virtual machines were connected directly through a virtual Ethernet pair (veth). We used the `tc` commands to regulate the bandwidth of the link in-between by randomly selecting a bandwidth from a set of values: 1 Mbps, 10 Mbps, 100 Mbps, and 1 Gbps.

We changed the bandwidth every second or every 100 ms and measured the average TCP throughput at the corresponding time intervals. Fig. 4.8 shows the results from a randomly chosen time period during the experiment. The left plot shows the results for changes at one-second interval. The average TCP throughput responds well to the bandwidth changes, except for a few instances (at time 33 and 36 seconds) when the bandwidth is drastically reduced from 1 Gbps to 1 Mbps. `tc` uses token buckets for regulating the packet transmission over the link; the higher than expected throughput is probably due to the backlog. The right plot of Fig. 4.8 shows the results for changes at

100 ms intervals. The TCP throughput does not seem to track the bandwidth changes as well as in the previous case. This means that regulating the bandwidth at the 100 ms time scale may introduce nontrivial inaccuracies.

4.4.3 Real Traffic Actuation

Base on last experiment results, we see `tc` mechanism can regulate the bandwidth very well on every 1 second time interval. In this experiment, we want to validate the accuracy of our queuing model in together with the robustness of traffic control. Our goal is to find out whether target flow in Mininet will reflect corresponding cross-traffic effect, either from background traffic in simulation or application traffic in other Mininet instances.

We still used a dumbbell model in this test, which consists of six end hosts connected by two routers. The communication between either two hosts from each side is sharing a bottleneck link. The bandwidth and the propagation delay are configured to be the same as in the previous experiments. We designated two end hosts (at the top on either side) to be the emulated hosts. Correspondingly, the downscaled topology that consists of two hosts connected by one three-link network pipe is instantiated in the Mininet instance. The other four end hosts served as simulated hosts to provide background traffic. The model is shown in Fig. 4.9.

In the experiment, we directed three TCP flows generated by a client/server application. *Flow 1* is a long-live emulated flow, which is running from system start until the very end, around 30 seconds. We artificially set a large number as traffic demand instead of accepting online-demand from emulator so that we can isolate test the interaction from simulation to emulation. The other two flows are all simulated flows. *Flow 2* contains five simultaneous TCP sessions each transferring 0.5 MB of data and all starting at 10 seconds. At 20 seconds, *Flow 3* starts with another 5 TCP sessions, each transferring 2 MB of data. We intentionally make *Flow 1* will share a bottleneck with another flow at

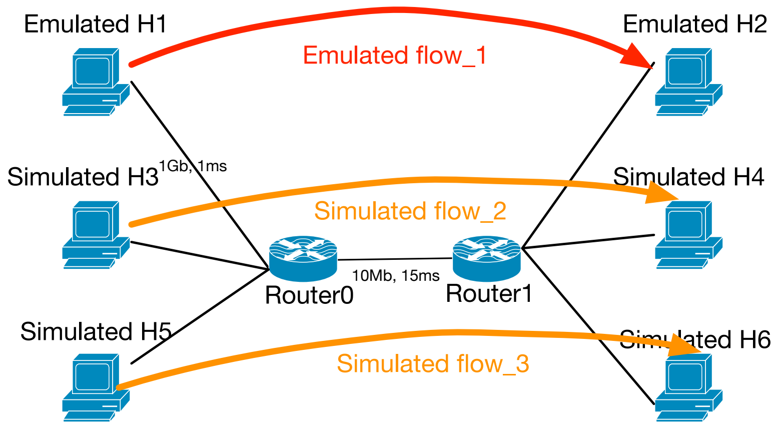


Figure 4.9: Model for real traffic regulation.

a different time. We want to see the fluctuation to *Flow 1* that is caused by the arrival and departure of its competent, *Flow 2, and Flow 3*, can be reflected on the emulation side at the correct time. In this case, there are two things can be validated through this setup: one is the correctness of the queuing model; the other is no significant delays of synchronization and bandwidth precise control.

Left plot of Fig. 4.13 shows the measured throughput for *Flow 1* at each second from Mininet. We also run `iperf` between the emulated hosts in Mininet so that we have the bandwidth during the experiment. The plot shows that the throughput of the emulated flow jumps accordingly both at 10 seconds and 20 seconds, which is the prompt reaction for the join of *Flow 2 and 3*. From the plot, we can see *Flow 1* almost occupied the total bandwidth of bottleneck link at around 15 seconds, at which *Flow 2* finished its downloading. Table 4.1 lists the instantaneous transmission rate of *Flow 1* both from Mininet and the simulator in a two-second interval, to further investigate the precision of bandwidth control. We also record the average throughput from both sides during the experiment. The throughput from Primex is 7.44 Mbps and from Mininet is 7.62 Mbps, which means the real traffic in emulation can also reflect the average behavior as it is in-situ transparently in the simulated network.

Table 4.1: Instantaneous throughput comparison in traffic regulation.

Interval (s)	Simulation Rate	Emulation Rate
0-2	10	9.54
2-4	9.4	8.98
4-6	9.9	9.45
6-8	10	9.55
8-10	10.2	9.73
10-12	5.3	5.08
12-14	5.8	5.56
14-16	9.3	8.88
16-18	9.2	8.79
18-20	9.8	9.36
20-22	5.1	4.89
22-24	5.8	5.56
24-26	5.2	4.98
26-28	6.1	5.85
28-30	5.2	4.98

4.4.4 Union System Test

As have already tested the feasibility of interaction independently from emulation to simulation and from simulation to emulation, our next move is to conduct preliminary experiments to validate the accuracy of the whole system. We compare the receiving data sequence history produced by the real applications running in Mininet with those produced by similar applications running in simulation. We first study *mininet-symbiosis* with one emulated flow running on one Mininet instance, while interacting with several other simulated flows. Then the experiment is extended to coordinate separate Mininet instances, each representing a different set of application flows that share the overlapping paths.

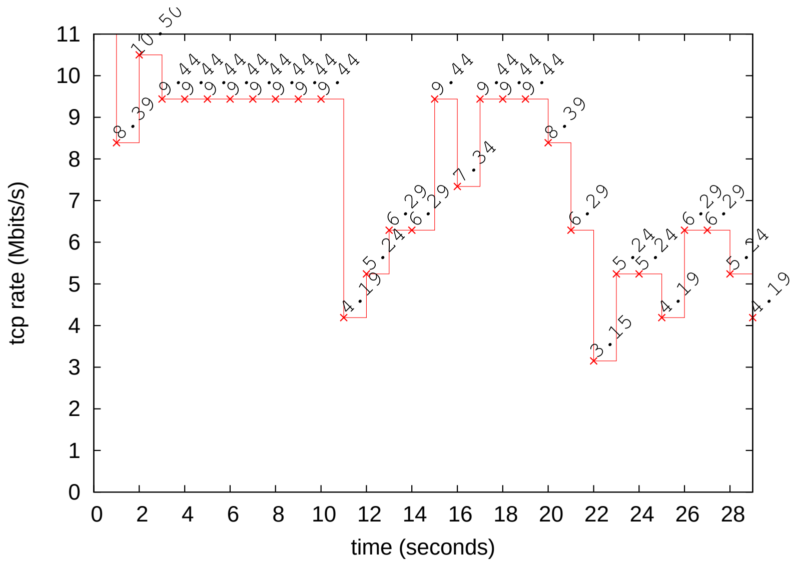


Figure 4.10: Throughput History in Mininet.

Single Mininet Instance

In the first experiment, we reused the model in the previous traffic control experiment 4.9, and set up the similar traffic pattern. We complement the last experiment with regenerating flow in the counterpart simulation application according to the real-time demand received from Mininet, to close the loop. Although 100ms time interval has better accuracy for reproducing traffic, we use 1 second as synchronization interval to balance the performance for `tc` traffic control.

Fig. 4.11 plots the sequence number (in bytes) of the received TCP segments by the emulated host over time. We show the results from both Mininet and simulation. From the figure, the sequence number history from two systems is very similar. We also measure the average TCP rate for the emulated flow from both side, which is 7.40 Mbps and 7.68 Mbps. Despite some difference, the outcome from this experiment basically demonstrates the feasibility of applying our symbiotic approach with Mininet.

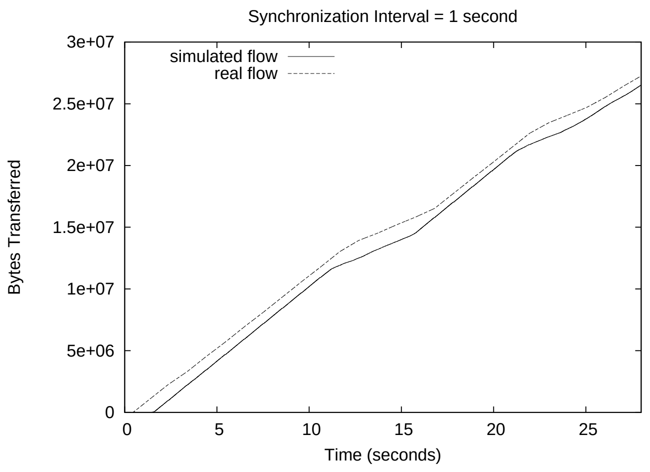


Figure 4.11: Average throughput of a single application flow in union system test.

Multiple Mininet Instances

Next, we perform the preliminary experiment of validating the entire system that coordinates multiple Mininet instances, each representing a different set of the overlapping network flows. We try two Mininet instances at the beginning. In this experiment, we still used the dumbbell model, similar as shown in Fig. 4.2. We used the same value to configure the bandwidth and the propagation delay, either for bottleneck link, or for spoke link. However, there are two set of application flows `iperf` running from the host of one side to the host of the other side. We instantiated each application on one Mininet running separately on a machine (or a VM), although their flows are sharing the bottleneck link. We want to investigate the interaction between different application flows can be accurately expressed in our system.

In the experiment, two set of TCP flows are instantiated by `iperf` in both Mininet instances. *Emulated flow 1* is traversing between H1 and H3 from 0 seconds to 21 seconds; *emulated flow 2* is between H2 to H4, from 15s second to 30 seconds. In this case, both target flows are overlapping at bottleneck link of sometime during the system running.

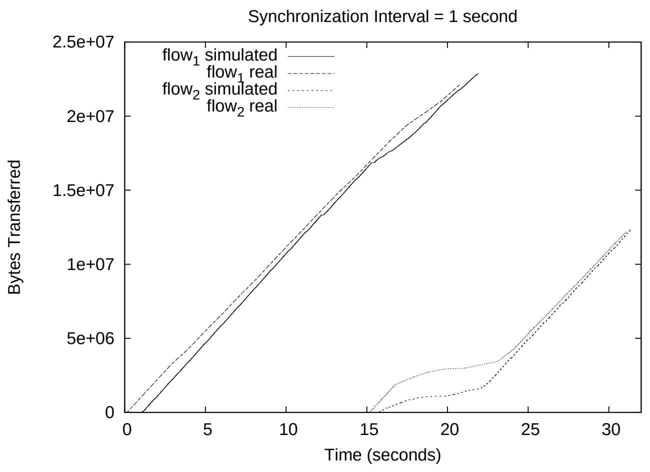


Figure 4.12: Received data history of real flows in two separate Mininets.

Table 4.2: Average throughput of two real flows in dummbell model.

	Simulation Throughput	Emulation Throughput
Emulated Flow 1	8.78	8.59
Emulated Flow 2	6.32	6.12

The results are shown in below. Left Fig. 4.12 is the sequence number for both flows from the receivers during the experiment. Again, we show the results from two Mininet instances, along with the results from simulation. From the figure, we see the sequence curves for *emulated flow 1* are indistinguishable from each other. The real traffic curve is catching up with simulated traffic curve for *emulated flow 2* while there is a gap at the beginning of its join. We speculate the reason for that is `tc` requires a certain time for sudden changing the bandwidth from a very large value to a very small value. Table 4.2 shows the average throughput over the experiment time between Mininet and simulation of both flows. The overall behavior of two system matches quite well. The differences are around 2%-3%.

To further investigate whether *mininet-symbiosis* can capture the interaction between different real flows accurately and timely, we conduct another test. The model is like 4.9, which has the same setting of paramenters. We directed three TCP flows with `iperf` (using TCP Reno), one from each host on one side of the dumbbell to a different host

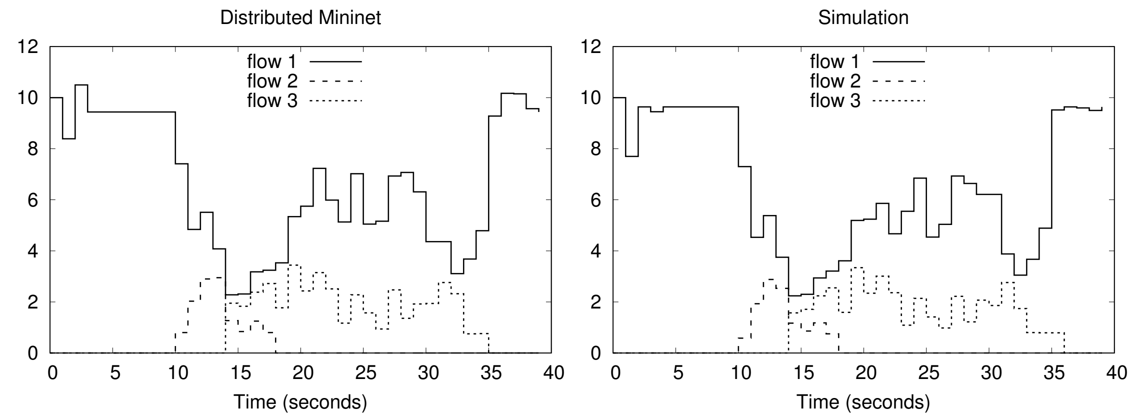


Figure 4.13: Comparing TCP throughput (in Mb/s) of the three flows from distributed Mininet vs. simulation.

on the other side. The first flow was a long-lived flow starting from the beginning of the experiment. The second flow started from 10 seconds and lasted for 8 seconds. The third flow started from 14 seconds and ended at 35 seconds. In this experiment, for distributed emulation, we instantiated three Mininet instances, one for each flow. For comparison, we created the same scenario in simulation. Fig. 4.13 shows that the throughput from distributed emulation (reported by `iperf`) match well with the simulation output.

4.5 Preliminary Experiment of Distributed Emulation with Symbiosis

The validation tests basically show the accuracy of *mininet-symbiosis*, which applies the symbiotic approach to integrating Mininet and simulation. As tried to coordinate multiple Mininet instances, our system has the capability of doing distributed emulation with symbiosis. This approach deals with the primary problem of Mininet for its capacity limitation. We conduct a preliminary experiment that consists of a mid-size network loaded with several target-flows simultaneously, to show the capability of doing distributed emulation with symbiosis. This test gives a good example of using the system to study the behavior of peer to peer applications, which contains multiple application traffic between

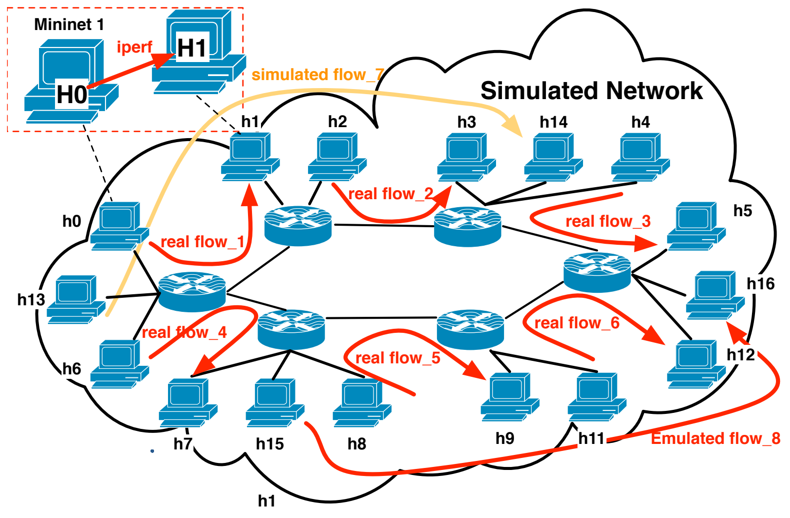


Figure 4.14: Ring model.

different hosts at the same time. We also want to investigate whether our approach can be the fundamental way to improve the scalability of physical experiments.

We used the ring model in this experiment, in which case the network contains 16 hosts and 6 routers. Although the size is not very large, the ring model can be expanded in the future. We set the bandwidth of all links between two routers to be 100 Mbps, which is ten times of the bandwidth we used in our validation tests. All other links are still to be 1Gbps. The propagation delays are configured as 5 millisecond and 1 millisecond accordingly. We used `iperf` to generate short-lived six real TCP flows; the source and destination of each flow are two random hosts that connected by two neighbored routers. Each `iperf` flow is generated from a single Mininet instance running on a separate machine. We also set up one simulated flow and one long-lived emulated flow across two bottlenecks to compete for the bandwidth with short-live emulated flows, to make the experiment more interesting. Long-lived real traffic is also coming from `iperf` application. Since multiple real flows are flowing through, our approach amortizes the execution of each on individual machines with coordinating them on the simulator. In this case,

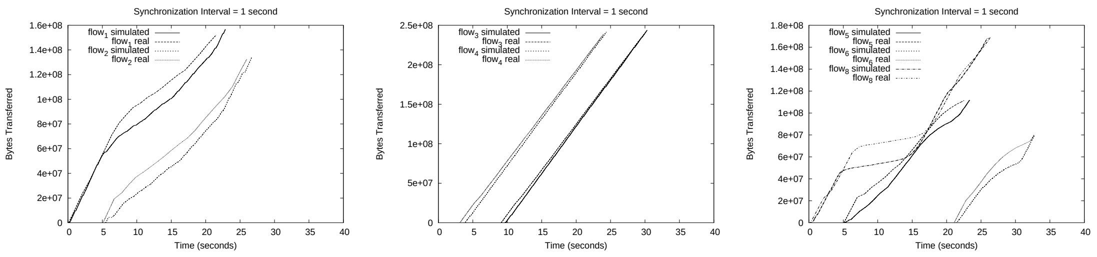


Figure 4.15: Receiving data for all real flows.

Table 4.3: Flow duration in ring model experiment.

	Start Time	End Time	Simulated or Emulated	Application
Flow 1	0	20	emulated	iperf
Flow 2	5	25	emulated	iperf
Flow 3	0-10	20-30	emulated	iperf
Flow 4	0-10	20-30	emulated	iperf
Flow 5	5	20	emulated	iperf
Flow 6	20	30	emulated	iperf
Flow 7	0	10-20	simulated	5 session, each of 20MB
Flow 8	0	25	emulated	iperf

the total amount of traffic handled in the experiment can be increased significantly. The details of the model and the flows are shown in Fig. 4.14.

We named one-hop emulated flows from flow_1 to flow_6, two-hop simulated flow as flow_7, two-hop emulated flow as flow_8. For the experiments, we designated the duration time for each flow specifically as table 4.3. From Fig. 4.14, we can see flow_1 and flow_2 share bottleneck with flow_7, flow_5 and flow_6 share bottleneck with flow_8. We set up the experiment on seven computing nodes from Apt Cluster of CloudLab [Clo]. We used the machines at CloudLab [Clo] for all our experiments. Each of these machines is equipped with two eight-core Intel Xeon E5-2450 2.1 GHz processors and 16 GB memory; they are connected by 10 Gbps Ethernet. The PRIME simulator and six Mininet instances are running distributedly on separate machines. We let the emulated flows start several seconds after the simulation begins because the simulator and the Mininet instances require some to synchronize with each other.

Table 4.4: Average throughput comparison of six real flows in ring model experiment.

	Simulation Throughput	Emulation Throughput	Error
Flow 1	56.8	56.8	0.0%
Flow 2	51.2	50.7	1.0%
Flow 3	95.3	92.0	3.6%
Flow 4	94.3	91.6	2.9%
Flow 5	48.5	51.1	5.1%
Flow 6	55.3	55.2	0.2%
Flow 8	51.8	51.7	0.2%

Fig. 4.15 shows the TCP sequence number along time from the receiver of all `iperf` flows created in Mininet instances. In the figures, we use the results from the simulation as comparison. We show the results separately in three groups. From the results, we can see flow_3 and flow_4 have the best matches. The other four one-hop emulated flows and the two-hop emulated flow have similar curve shape of their simulation counterparts, whereas with some differences. We think the difference is mostly caused by the simulator could not synchronize perfectly with different Mininet instances simultaneously. We also list the average throughput for all emulated flows during the simulation in Table 4.4. We observe the accuracy are good for average behaviors of all the flows. The error never exceeds 5.1%.

4.6 Case Study

Denial of service (DoS) attacks prevents the target computer from responding quickly to its legitimate users' traffic, or not at all. Shrew is a specific attack pattern where the attacher sends bursts of data at a regular interval to an over-committed bottleneck link [KK03]. When the attack bursts occur at intervals that synchronize with the minimum retransmission timeout (RTO) of legitimate TCP connections sharing the bottleneck link, they can trigger TCP timeouts and consequently strangle the throughput of those connections. Since the average traffic rate of a shrew attack is low, it can be difficult to

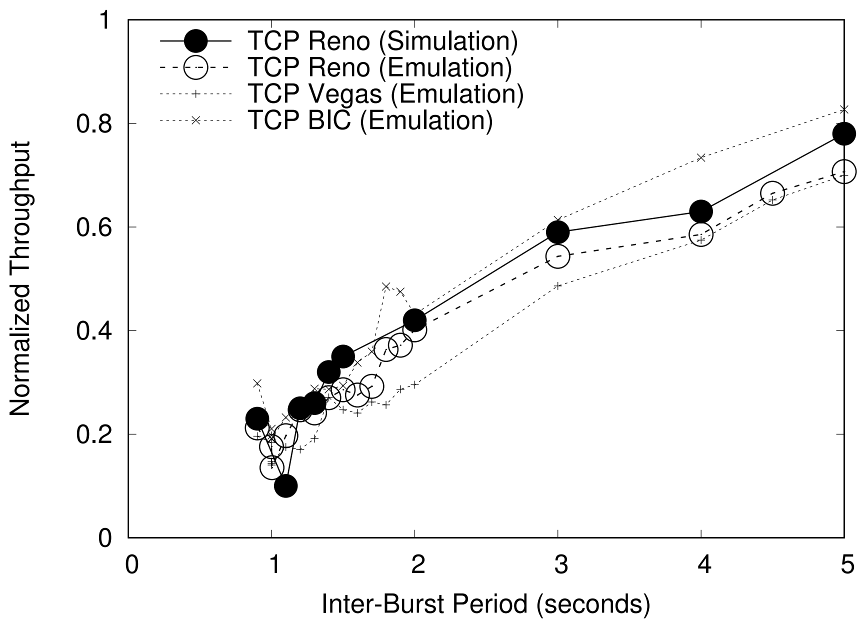


Figure 4.16: The effect of the Shrew attack.

be detected. In this section, we use our distributed Mininet to study the behavior of the Shrew attack.

To establish the baseline, we start with the same experiment setup with a simple topology as in [KK03]. There is one pair of good sender and receiver (the victim flow) and another pair of bad sender and receiver (the attack flow); they share the same bottleneck link. The good pair are separated by two routers, and the bad pair by three routers. The shared bottleneck link has 10 Mb/s bandwidth and 20 ms delay. All the other links have 100 Mb/s bandwidth and 2 ms delay. We ran one Mininet instance to emulate the good data transfer (using `iperf`) and simulate the attack flow using UDP. We set the burst rate to be 10 Mb/s and the length of each burst to be 100 ms. We ran experiments with different inter-burst period (the time between consecutive bursts) from 0.9 to 5 seconds.

The results are shown in Fig. 4.16. The y-axis is the normalized throughput, which is the throughput of the victim flow divided by the bandwidth of the bottleneck link. Our results, marked as “TCP Reno (Emulation)”, are comparable with “TCP Reno (Simulation)", the results reported in the original paper [KK03]. We also tried other TCP versions (Vegas and BIC are shown in the plot); we got similar results. As expected, we see that the Shrew attack can significantly lower the throughput of the victim flow (in this case when the inter-burst period is at around 1 second).

We conducted another experiment using the dumbbell topology, where we have five pairs of good senders and receivers and one pair of bad sender and receiver. We set the bandwidth of the bottleneck link to be 100 Mb/s and the other links to be 1 Gb/s. We set the victim flows to have different round-trip times (RTTs): 40, 80, 160, 280 and 360 ms. The attack flow has an RTT of 440 ms. The attack flow has the same burst and length as in the previous example. We fix the inter-burst period to be 1.0003 seconds.

We use distributed emulation with six Mininet instances, one for each flow. We compare the results obtained from the distributed Mininet with those from running a single Mininet instance. In Fig. 4.17, one can see that running a single Mininet instance, the Shrew DoS attack basically has no effect on the throughput of the flows. In fact, it generates incorrect results: the flow with the smallest RTT got the whole share of the throughput. Using our distributed Mininet, we observe that not only the aggregate throughput is reduced by the DoS attack, but also the flows react differently: the throughput degrades more significantly for flows with higher RTTs.

4.7 Conclusion

Symbiotic simulation provides a promising tradeoff, by combining the emulation testbeds, which can feature a more realistic environment for running network applications, and simulation, which can provide more flexible, large, and complex network scenarios. In particular, we outline a specific design of combining instances of a popular network emulator, called Mininet, with a real-time simulator, called PrimoGENI. We provide a detailed account on the use of low-level mechanisms for implementing the symbiotic approach in the

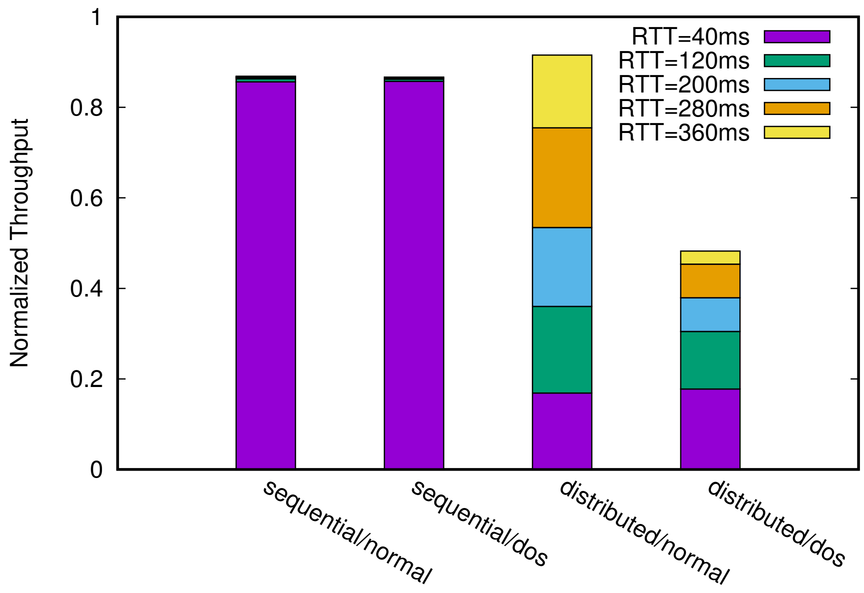


Figure 4.17: Sequential vs. distributed Mininet runs.

Linux environment. With this specific *mininet-symbiosis*, one can use Mininet to directly run applications on virtual machines and software switches, with network connectivity represented by detailed simulation at scale.

We also present a distributed emulation method using a symbiotic approach. In our approach, the simulator acts as a coordinator for the distributed emulation instances by capturing the effect of contention among the network flows potentially belonging to different distributed instances. Our approach provides a novel method for partitioning the virtual network among the emulation instances and can be used in accordance with the tradition spatial decomposition method. Through experiments using a distributed Mininet implementation, we show that the symbiotic approach can generate accurate results, and it can be readily used in studies involving high traffic load scenarios.

CHAPTER 5

CONCLUSIONS

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

5.1 Summary

The focus of this dissertation is to design a hybrid system that can conduct distribute at-scale network studies and experiments on diverse cyber-infrastructure. Specifically, we addressed the following problems:

1. **Problem:** make PDES scalable regardless of models and running platforms.

   Solution: a self-adaptive synchronization for PDES.

   The algorithm we designed is called hierarchical composite synchronization. The approach is extended from composite synchronization to avoid performance pitfalls of two traditional synchronization methods in parallel simulation. It is designed to address the discrepancy in the communication and synchronization cost for shared-memory multiprocessor multicore machines and distributed-memory machines. The synchronization can be tailored to exploit the parallelism of computing platforms that have disparate architectures. We implement the method in a minimalistic parallel simulator, called *MiniSSF*. Without knowing the resources type beforehand, one can still self-configure the synchronization automatically to achieve optimal performance while running the simulation. It is an expressive and flexible parallel simulator for different models and is able to run transparently on multiple supercomputers by harnessing their parallel capabilities to cope with large-scale models. We investigate the performance of *MiniSSF* through experiments on several different supercomputers.

2. **Problem:** provide a generalized system to validate design and implementation at-scale without loosing realism.

   Solution: distributed emulation with simulation symbiosis.

   We apply an existing symbiotic approach that effectively combine network emulation with simulation to a specific emulator–Mininet. One can use Mininet to directly run applications on virtual machines and software switches, with network connectivity represented by detailed simulation at scale. We present a distributed emulation method using a symbiotic approach. In our approach, the simulator acts as a coordinator for the distributed emulation instances by capturing the effect of contention among the network flows potentially belonging to different distributed instances. With *distributed mininet-symbiosis*, we can conduct hybrid at-scale experiments distributedly with any cyber-testbeds and test applications and algorithms easily with various system configurations and design parameter.

5.2 Future Directions

The research presented in this dissertation can be extended in a few directions. We emphasize the extension of *mininet-symbiosis* because it has the potential to be an effective testbed to validate design and implementation issues of future Internet applications and protocols.

To improve our parallel simulator *MiniSSF* would be necessary for increasing the scalability of the hybrid testbed. *MiniSSF* could be improved in the following ways:

1. As for hierarchical composite synchronization, although we have proposed a performance model to predict its behavior 3, the model is not good enough to speculate optimal configurations of the algorithm. we would like to continue refining the performance model for more accurate predictions, and hopefully set it indepen-

dent from the direct measurements on the specific target platform. The hierarchical composite algorithm can also be extended to include more distinct levels to reflect the difference in the cost of communicating, e.g., between machines at different cabinets or on the same racks, between processors on different processor boards or on the same compute card, and between the cores on different processors or within the same chip. We also plan to extend the hybrid approach for topology-inspired synchronization schemes.

2. Immediate future work of the simulator itself is to conduct more experiments to fine-tune the simulator's performance on various high-performance computing architectures.

3. Automatic configuration for the simulator could also be improved to make it adapt to the runtime environment.

There are several things that we would like to explore with the *mininet-symbiosis* in the future work. Possible directions include the following:

1. Our symbiotic approach is currently building on a specific simulator and a specific emulator. It is designed for validating our symbiotic idea, as well as for targeting a specialized class of applications and protocols. However, the idea should be applied to experimental testbeds in general. We will isolate the utility modules as plug-ins for most commonly used emulators and simulators. It will also be further extended to include physical testbeds.

2. We would like to apply *mininet-symbiosis* to studying bandwidth-intensive OpenFlow applications, which would otherwise be difficult to realize in the traditional simulation or emulation testbeds.

3. In particular for the distributed Mininet with symbiosis, future works are:

- We would like to first integrate our symbiotic approach with the traditional spatial decomposition. In this case, a robust partitioning algorithm is needed to be able to handle different scenarios.
- Our current method requires that significant flows be identified during experiment configuration. This can be an unnecessary burden if the system can dynamically identify these flows and create network pipes on demand.
- Our current design has but one centralized simulation controller. A distributed approach is needed to avoid the potential bottleneck for a large number of emulation instances.
- We would like to explore other more efficient simulation abstractions (such as fluid models) which can further reduce the cost of the controller.

BIBLIOGRAPHY

[ADHK08] J. Ahrenholz, C. Danilov, T.R. Henderson, and J.H. Kim. Core: A real-time network emulator. In *Proceedings of the IEEE Military Communications Conference (MILCOM)*, pages 1–7, 2008.

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

[Aya89] Rassul Ayani. A parallel simulation scheme based on the distance between objects. *Proceedings of the 1989 SCS Multiconference on Distributed Simulation*, 21(2):113–118, 1989.

[BCH09] David W. Bauer, Jr, Christopher D. Carothers, and Akintayo Holder. Scalable Time Warp on Blue Gene supercomputers. In *Proceedings of the 23rd Workshop on Principles of Advanced and Distributed Simulation (PADS'09)*, pages 35–44, 2009.

[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.

[BMT$^+$98] Rajive Bagrodia, Richard Meyer, Mineo Takai, Yu an Chen, Xiang Zeng, Jay Martin, and Ha Yoon Song. PARSEC: a parallel simulation environment for complex systems. *IEEE Computer*, 31(10):77–85, 1998.

[Bro88] R. Brown. Calendar queues: a fast o(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31:1220–1227, 1988.

[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.

[BTA+99] Lokesh Bajaj, Mineo Takai, Rajat Ahuja, Ken Tang, Rajive Bagrodia, and Mario Gerla. GloMoSim: a scalable network simulation environment. Technical Report 990027, Department of Computer Science, UCLA, 1999.

[CBP00a] 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.

[CBP00b] 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, 2000.

[Cha] Chameleon - A configurable experimental environment for large-scale cloud research. `https://www.chameleoncloud.org/`.

[cla] Clang: a C language family frontend for LLVM. `http://clang.llvm.org/`.

[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.

[Clo] CloudLab. `https://www.cloudlab.us/`.

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

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

[CP10] Christopher D. Carothers and Kalyan S. Perumalla. On deciding between conservative and optimistic approaches on massively parallel platforms. In *Proceedings of the 2010 Winter Simulation Conference (WSC'10)*, pages 678–687, 2010.

[CPF99] Christopher D. Carothers, Kalyan S. Perumalla, and Richard M. Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM Transactions on Modeling and Computer Simulation*, 9(3):224–253, 1999.

[CR10] Marta Carbone and Luigi Rizzo. Dummynet revisited. *SIGCOMM Comput. Commun. Rev.*, 40(2):12–20, April 2010.

[CS89] K. M. Chandy and R. Sherman. The conditional event approach to distributed simulation. *Proceedings of the 1989 SCS Multiconference on Distributed Simulation*, 21(2):93–99, 1989.

[CT90] Wentong Cai and Stephen J. Turner. An algorithm for distributed discrete-event simulation—the "carrier null message" approach. *Proceedings of the 1990 SCS Multiconference on Distributed Simulation*, 22(1):3–8, 1990.

[DDD14] DDDAS. Dynamic Data-Driven Application Systems Info Cybernetics, 2014. `http://www.dddas.org/`. Last access: August 2014.

[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 1994 Winter Simulation Conference (WSC'94)*, pages 1332–1339, 1994.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI'04)*, 2004.

[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.

[EL13] Miguel A. Erazo and Jason Liu. Leveraging symbiotic relationship between simulation and emulation for scalable network experimentation. In *Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS)*, pages 79–90, 2013.

[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.

[ERL15] Miguel A. Erazo, Rong Rong, and Jason Liu. Symbiotic network simulation and emulation. *ACM Trans. Model. Comput. Simul.*, 26(1):2:1–2:25, June 2015.

[ESn] ESnet: Energy Sciences Network. http://www.es.net/.

[eXp] eXpressive Internet Architecture (XIA) Project. http://www.cs.cmu.edu/~xia/.

[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.

[FLPU02] R. Fujimoto, D. Lunceford, E. Page, and A. M. Uhrmacher. Grand challenges for modeling and simulation. Technical Report 350, Schloss Dagstuhl, 2002.

[Fos95] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley, 1995.

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

[Fuj90] Richard M. Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33(10):30–53, 1990.

[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.

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

[GENb] GENI Racks. http://groups.geni.net/geni/wiki/GENIRacksHome.

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

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

[HHJ+12] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible network experiments using container-based emulation. In *CoNEXT*, pages 253–264, 2012.

[HN93] Philip Heidelberger and David Nicol. Conservative parallel simulation of continuous time Markov chains using uniformization. *IEEE Transactions on Parallel and Distributed Systems*, 4(8):906–921, 1993.

[HRS+08] Mike Hibler, Robert Ricci, Leigh Stoller, Jonathon Duerig, Shashi Guruprasad, Tim Stack, Kirk Webb, and Jay Lepreau. Large-scale virtualization in the emulab network testbed. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 113–128, Berkeley, CA, USA, 2008. USENIX Association.

[Int] Internet2. `http://www.internet2.edu/`.

[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.

[KK98] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20:359–392, 1998.

[KK03] Aleksandar Kuzmanovic and Edward W. Knightly. Low-rate tcp-targeted denial of service attacks: The shrew vs. the mice and elephants. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '03, pages 75–86, New York, NY, USA, 2003. ACM.

[LC03] Xin Liu and Andrew A. Chien. Traffic-based load balance for scalable network emulation. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing (SC'03)*, 2003.

[LHM10] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM Workshop on Hot Topics in Networks*, pages 19:1–19:6, 2010.

[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.

[Liu13] Jason Liu. Real-time scheduling of logical processes for parallel discrete-event simulation. In *Proceedings of the 2013 Winter Simulation Conference (WSC'13)*, pages 2959–2971, 2013.

[LL14] Ting Li and Jason Liu. Cluster-based spatiotemporal background traffic generation for network simulation. *ACM Trans. Model. Comput. Simul.*, 25(1):4:1–4:25, 2014.

[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.

[LLV$^+$09] Jason Liu, Yue Li, Nathanael Van Vorst, Scott Mann, and Keith Hellman. A real-time network simulation infrastructure based on OpenVPN. *Journal of Systems and Software*, 82(3):473–485, 2009.

[LMAR15] J. Liu, C. Marcondes, M. Ahmed, and R. Rong. Toward scalable emulation of future internet applications with simulation symbiosis. In *2015 IEEE/ACM 19th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pages 68–77, Oct 2015.

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

[LN01] Jason Liu and David M. Nicol. Learning not to share. In *Proceedings of the 15th Workshop on Parallel and Distributed Simulation (PADS'01)*, pages 46–55, 2001.

[LNPP99] Jason Liu, David Nicol, Brian Premore, and Anna Poplawski. Performance prediction of a parallel simulator. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation (PADS'99)*, pages 156–164, 1999.

[LR12] Jason Liu and Rong Rong. Hierarchical composite synchronization. In *Proceedings of the 2012 Workshop on Principles of Advanced and Distributed Simulation (PADS'12)*, pages 3–12, 2012.

[Lub88] Boris D. Lubachevsky. Bounded lag distributed discrete event simulation. *Proceedings of the 1988 SCS Multiconference on Distributed Simulation*, 19(3):183–191, 1988.

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

[LYN$^+$05] Jason Liu, Yougu Yuan, David M. Nicol, Robert S. Gray, Calvin C. Newport, David Kotz, and Luiz Felipe Perrone. Empirical validation of wireless models in simulations of ad hoc routing protocols. *SIMULATION: Transactions of The Society for Modeling and Simulation International*, 81(4):307–323, April 2005.

[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.

[MAB$^+$08] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, 2008.

[MBI$^+$14] Ali Munir, Ghufran Baig, Syed Muhammad Irteza, Ihsan Ayyub Qazi, Alex Liu, and Fahad Dogar. Friends, not foes - synthesizing existing transport strategies for data center networks. In *Proceedings of the 2014 ACM SIGCOMM Conference (SIGCOMM)*, 2014.

[MET] METIS. `http://glaros.dtc.umn.edu/gkhome/views/metis`.

[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'01)*, 2001.

[Mob] MobilityFirst Future Internet Architecture Project. `http://mobilityfirst.winlab.rutgers.edu/`.

[Nam] Named Data Networking (NDN) Project. `http://www.named-data.net/`.

[NEB] NEBULA Project. `http://nebula.cis.upenn.edu/`.

[Nic88] David M. Nicol. Parallel discrete-event simulation of FCFS stochastic queueing networks. *ACM SIGPLAN Notices*, 23(9):124–137, 1988.

[Nic93] David M. Nicol. The cost of conservative synchronization in parallel discrete event simulations. *Journal of the ACM*, 40(2):304–333, 1993.

[Nic96] David M. Nicol. Principles of conservative parallel simulation. In *Proceedings of the 1996 Winter Simulation Conference (WSC'96)*, pages 128–135, 1996.

[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.

[NJZ11] David M. Nicol, Dong Jin, and Yuhao Zheng. S3F: the scalable simulation framework revisited. In *Winter Simulation Conference*, pages 3288–3299, 2011.

[NL97] David M. Nicol and Jason Liu. The dark side of risk (what your mother never told you about Time Warp). In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation (PADS'97)*, pages 188–195, 1997.

[NL02] David M. Nicol and Jason Liu. Composite synchronization in parallel discrete-event simulation. *IEEE Transactions on Parallel and Distributed Systems*, 13(5):433–446, 2002.

[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`.

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

[ovs] Open vSwitch. `http://openvswitch.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.

[Per] Kalyan S. Perumalla. μsik - a micro-kernel for parallel/distributed simulation systems. In *Proceedings of the 19th Workshop on Parallel and Distributed Simulation (PADS'05)*, pages 59–68.

[Per07] Kalyan S. Perumalla. Scaling time warp-based discrete event execution to 104 processors on a Blue Gene supercomputer. In *Proceedings of the 4th International Conference on Computing Frontiers*, pages 69–76, 2007.

[Pla] `http://www.planet-lab.org/`.

[RAT93] Hassan Rajaei, Rassul Ayani, and Lars-Erik Thorelli. The local Time Warp approach to parallel simulation. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation (PADS'03)*, pages 119–126, 1993.

[RBZ$^+$14] Arup Raton Roy, Md. Faizul Bari, Mohamed Faten Zhani, Reaz Ahmed, and Raouf Boutaba. Dot: Distributed openflow testbed. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 367–368, 2014.

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

[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.

[RHL14] Rong Rong, Jiang Hao, and Jason Liu. Performance study of a minimalistic simulator on xsede massively parallel systems. In *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*, XSEDE '14, pages 15:1–15:8, 2014.

[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.

[SS89] Wen-King Su and C. L. Seitz. Variants of the Chandy-Misra-Bryant distributed discrete-event simulation algorithm. *Proceedings of the 1989 SCS Multiconference on Distributed Simulation*, 21(2):38–43, 1989.

[SSS+02] Boleslaw K. Szymanski, Adnan Saifee, Anand Sastry, Yu Liu, and Kiran Madnani. Genesis: a system for large-scale parallel network simulation. In *Proceedings of the 16th Workshop on Parallel and Distributed Simulation (PADS 2002)*, pages 89–96, May 2002.

[TUM+13] Hajime Tazaki, Frédéric Uarbani, Emilio Mancini, Mathieu Lacage, Daniel Camara, Thierry Turletti, and Walid Dabbous. Direct code execution: Revisiting library OS architecture for reproducible network experiments. In *CoNEXT*, pages 217–228, 2013.

[VEL11] Nathanael Van Vorst, Miguel Erazo, and Jason Liu. PrimoGENI: Integrating real-time network simulation and emulation in GENI. In *Proceedings of the 2011 Workshop on Principles of Advanced and Distributed Simulation (PADS'11)*, 2011.

[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.

[WDS+14] P. Wette, M. Draxler, A. Schwabe, F. Wallaschek, M.H. Zahraee, and H. Karl. Maxinet: Distributed emulation of software-defined networks. In *Proceedings of the 2014 IFIP Networking Conference*, pages 1–9, 2014.

[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.

[xse] XSEDE: Extreme Science and Engineering Discovery Environment. `http://www.xsede.org/`.

[XUSC99] Z. Xiao, B. Unger, R. Simmonds, and J. Cleary. Scheduling critical channels in conservative parallel discrete event simulation. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation (PADS'99)*, pages 20–28, 1999.

[YKH+01] Tao Ye, Shivkumar Kalyanaraman, David Harrison, Biplab Sikdar, Bin Mo, Hema Tahilramani, Ken Vastola, and Boleslaw Szymanski. Network management and control using collaborative on-line simulation. In *Proceedings of the IEEE International Conference on Communications (ICC'01)*, 2001.

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

[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

RONG RONG

December, 1984	Born, Gansu, China
2002-2006	B.E., Software Engineering BeiHang University Beijing, China
2015	M.S, Computer Science Florida International University Miami, Florida
2009-2016	Doctoral Candidate Florida International University Miami, Florida

PUBLICATIONS AND PRESENTATIONS

T. Li, N. Van Vorst, R. Rong, J. Liu. *Simulation Studies of OpenFlow-Based In-Network Caching Strategies*. Proceeding of the 15th Communications and Networking Simulation Symposium (CNS'12), Pages 12:1-12:7, March 2012.

J. Liu, R. Rong. *Hierarchical Composite Synchronization*. 26th Workshop on Principles of Advanced and Distributed Simulation (PADS'12), Page 3-12, July 2012.

R. Rong, Hao. J, and J. Liu. *Performance Study of a Minimalistic Simulator on XSEDE Massively Parallel Systems*. Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment (XSEDE), Page 1-8, July 2014.

M. Erazo, R. Rong, J. Liu. *Symbiotic Network Simulation and Emulation*. ACM Transactions on Modeling and Computer Simulation (TOMACS), Volume 26, Issue 1, Page 1-25, December 2015.

J. Liu, C. Marcondes, M. Ahmed, R. Rong. *Toward Scalable Emulation of Future Internet Applications with Simulation Symbiosis*. 2015 IEEE/ACM 19th International Symposium on Distributed Simulation and Real Time Applications (DS-RT), Page 68-77, September 2015.

R. Rong, J. Liu. *Distributed Mininet with Symbiosis*. Submitted to 2017 IEEE International Conference on Communications(ICC), May 2017.

R. Rong. *Hierarchical Composite Synchronization*. Paper presented at PADS'12 Conference, Zhangjiajie, China, July 15-19, 2012.

R. Rong. *Performance Study of a Minimalistic Simulator on XSEDE Massively Parallel Systems*. Paper presented at XSEDE'14 Conference, Atlanta, GA, July 13-18, 2014.