9-1-2005

Improving Large-Scale Network Traffic Simulation with Multi-Resolution Models

Guanhua Yan
Dartmouth College

Improving Large-Scale Network Traffic Simulation with Multi-Resolution Models

Dartmouth Computer Science Technical Report TR2005-558

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Doctor of Philosophy

in

Computer Science

by

Guanhua Yan

DARTMOUTH COLLEGE

Hanover, New Hampshire

September, 2005

Examining Committee:

(chair) David Kotz

Sean W. Smith

M. Douglas McIlroy

David M. Nicol (UIUC)

Charles K. Barlowe, Ph.D.
Dean of Graduate Studies

Abstract

Improving Large-Scale Network Traffic Simulation with Multi-Resolution Models

Dartmouth Computer Science Technical Report TR2005-558
by
Guanhua Yan
Doctor of Philosophy in Computer Science
Dartmouth College, Hanover, NH
September 2005

Simulating a large-scale network like the Internet is a challenging undertaking because of the sheer volume of its traffic. Packet-oriented representation provides high-fidelity details but is computationally expensive; fluid-oriented representation offers high simulation efficiency at the price of losing packet-level details. Multi-resolution modeling techniques exploit the advantages of both representations by integrating them in the same simulation framework. This dissertation presents solutions to the problems regarding the efficiency, accuracy, and scalability of the traffic simulation models in this framework. The "ripple effect" is a well-known problem inherent in event-driven fluid-oriented traffic simulation, causing explosion of fluid rate changes. Integrating multi-resolution traffic representations requires estimating arrival rates of packet-oriented traffic, calculating the queueing delay upon a packet arrival, and computing packet loss rate under buffer overflow. Real time simulation of a large or ultra-large network demands efficient background traffic simulation. The dissertation includes a rate smoothing technique that provably mitigates the "ripple effect", an accurate and efficient approach that integrates traffic models at multiple abstraction levels, a sequential algorithm that achieves real time simulation of the coarse-grained traffic in a network with 3 tier-1 ISP (Internet Service Provider) backbones using an ordinary PC, and a highly scalable parallel algorithm that simulates network traffic at coarse time scales.

Acknowledgments

I want to give my utmost gratitude to Professor David Nicol, my research advisor, for his support and guidance on my thesis research. His broad knowledge assisted me through my graduate studies; his sharp insight helped me identify interesting research problems; his high standard was the driving force behind my thesis work. I look forward to collaborating with him in the future.

I would also like to thank my dissertation committee, Professor David Kotz, Professor Sean Smith, and Professor Doug McIlroy for many helpful suggestions and advice. Some of their comments on my thesis will motivate me for further exploration in the field of network simulation.

This dissertation was made possible only after many inspiring discussions with my colleagues, especially Michael Liljenstam, Jason Liu, Yougu Yuan, and Meiyuan Zhao. I thank Felipe Perrone for offering me to use his multiprocessor. I also want to express my gratitude to the people in the Computer Science Department at Dartmouth College, the staff in the Institute of Security Technology Studies, and the staff in the Coordinated Science Laboratory at University of Illinois, Urbana-Champaign. They offered me tremendous help throughout my graduate studies.

I want to thank all the friends with whom I spent my time in the last five years. They include Caie Yan, Dongsheng Zhang, Yaozhong Zou, Qing Feng, Shengyi Ye, Cheng Zhong, Feng Cao, Puxuan Dong, Yijin He, Xiang Li, Zhan Xiao, Qun Li, Guofei Jiang, and many many others.

I would like to express my greatest gratitude to my parents and my brothers for their love and support. Their encouragement has been a great source of inspiration in my life.

Funding

This research was supported in part by DARPA Contract N66001-96-C-8530, and NSF Grant CCR-0209144. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes. In addition this research program is also a part of the Institute for Security Technology Studies, supported under Award number 2000-DT-CX-K001 from the Office for Domestic Preparedness, U.S. Department of Homeland Security, Science and Technology Directorate. Points of view in this document are those of the author and do not necessarily represent the official position of the U.S. Department of Homeland Security or the Science and Technology Directorate.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The size of the Internet has undergone rapid growth over the past several decadses. When it was born in 1969, it had only four nodes, connected with 56kbps circuits [99]. But from the Internet domain survey done by *Internet Systems Consortium*[1], the number of nodes that are connected to the Internet has increased exponentially from 1.3 million in January 1993 to 317 million in January 2005. At the same time, the traffic volume carried by the Internet also increases at a high rate. In [110], it is estimated that the traffic on Internet backbones in U.S. has grown from 1.0 TB/month in 1990 to 80,000-140,000 TB/month in 2002. It is predicted that Internet traffic will continue to grow vigorously at a rate close to 100% per year [110][36]. In addition, although the Internet was designed for data communications, emerging applications such as Voice over IP(VoIP) and video-conferencing are providing increasingly versatile services to the end users.

The changes that have occurred to the Internet in the past several decades suggest its increasingly important role in people's daily routines. When its scale goes up, however, unprecedented challenges arise on many of its aspects. A few of them are briefly discussed as follows.

- **Congestion control.** TCP governs a dominant fraction of the current Internet traffic. The measurements on a backbone link show that 95% of the bytes, 90% of the packets and 80% of the flows attribute to the TCP protocol [25]. As the Internet continues to evolve, it will incorporate more and more high-bandwidth optical links and large-delay satellite and wireless links. In the context of high bandwidth-delay product networks, TCP becomes oscillatory and prone to instability, regardless of what queueing schemes are deployed in the network [87][60]. In addition, TCP is essentially an additive-increase-multiplicative-decrease (AIMD) protocol. Therefore, when a congestion signal(e.g., packet losses) is detected in a high bandwidth-delay network, TCP shrinks its congestion window size immediately as a result of its multiplicative-decrease policy, and it then has to suffer a large number of Round Trip Times(RTTs) before its congestion window ramps up to a high size because of its additive-increase policy. This may severely affect the TCP throughput in high bandwidth-delay product networks. All these issues raise significant performance concerns in the future Internet.

[1] http://www.isc.org

- **Routing.** Routing protocols in a packet-forwarding network like the Internet discover paths along which packets are directed to their destinations. Routing is an important means of traffic engineering, which deals with "*performance evaluation and performance optimization of operational IP networks*" [4]. The objectives of routing decisions can be manifold. They include balancing traffic load to prevent congestions, minimizing the amount of computation resource for packet processing, and satisfying the QoS(Quality of Service) requirements of end applications, and so on. Dynamic routing aiming to achieve these goals in an online fashion is particularly challenging [22]. One reason is that in a large, dynamic network such as the Internet, the knowledge of its up-to-date global state is hard to obtain at each router. On the other hand, the QoS requirements imposed by applications like video conference and Internet telephony are so diverse that it is *NP-hard* to satisfy some combinations of QoS constraints simultaneously [140].

  BGP(Border Gateway Protocol) [123] is the de facto standard protocol that currently governs the inter-domain Internet routing between *Autonomous Systems*(ASs). Although ubiquitous, BGP has severe problems if the Internet continues its exponential growth. First, route oscillations can happen because of the propagation of unstable route updates. The route flap dampening scheme, which is widely deployed on core BGP routers, is found to possibly "significantly exacerbate the convergence times of relatively stable routes [91]". Second, the experiments in [69] show that inter-domain routers may need tens of minutes to converge to a consistently stable state after a failure. Third, there is a concern regarding the scalability of BGP protocol because its routing table size has increased dramatically in the recent years [50]. As its result, packet forwarding becomes slower and more memory space are demanded to accommodate the expanding BGP routing table. Finally, because BGP was not designed to facilitate traffic engineering tasks, inter-domain traffic engineering is still at its infancy [4][33].

- **Security.** Over the last decade, the Internet has witnessed a surge of malicious attacks like worm and DDoS(Distributed Denial of Services) attacks. For instance, during the week of February 7th through 11th in 2000, many sites including Yahoo, Amazon, eBay and CNN became unreachable because of DDoS attacks [43]; more than 359,000 Microsoft IIS servers were infected by Code Red worm version 2 in less than 14 hours on July 19th, 2001 [97]; and the more recent SQL *Slammer* worm took only 10 minutes before infecting more than 90% of all the vulnerable machines [96]. The malware has caused enormous economic losses to the society. From the estimation by vnunet.com[2], the total cost of malware, including viruses, worms, and Trojans, totaled 166 billion dollars in 2004.

  The pressing threats from the increasingly rampant malware code pose significant challenges to the existing security architectures. For instance, conventional perimeter firewalls have been an important means of access control and protection from attacks on corporate computer networks for many years. However, the vastly expanded Internet connectivity gradually blurs the boundary between an enterprise Intranet and its outside network. Therefore, traditional "boundary firewalls" are losing their effectiveness as a growing number of hackers are able

[2] http://www.vnunet.com

to sneak such a single point of protection. On the other hand, distributed firewalls [52][7], proposed recently based on the "defense-in-depth" principle [45], are not a panacea, because they bring concerns over the potential vulnerabilities due to policy conflicts in middle- or large-sized enterprise networks [2].

- **Resource sharing.** As the Internet continues growing rapidly, there are vastly heterogeneous computing resources, such as processing power, storage, and communication bandwidth, that are scattered across multiple administrative domains. In order to facilitate efficient resource sharing, new computing platforms like peer-to-peer and Grid computing are deployed. The SETI@Home project [3], which searches for extraterrestrial intelligence by utilizing processing powers over the Internet, is such an example. Evaluating these platforms is a challenging task. For instance, a fundamental problem in designing a resource sharing system is its workload scheduler that can "*adapt to a wide spectrum of potential computational environments* [9]". Because such systems may span over multiple WANs(Wide Area Networks) geographically, dynamics in the underlying networks can severely impact the effectiveness and robustness of a resource scheduler. It is hard, though, to characterize the behavior of a large, dynamic network like the Internet, making it difficult to evaluate the performance of a workload scheduler.

In the above, we have only covered a few problems that need to be addressed when the Internet continues to grow. As the recent enthusiasm in networking research suggests, there are many others that still remain to be solved.

1.1 Motivations

Many methodologies, including theoretical analysis, measurements, and simulation, have been applied in investigating the large body of Internet-related problems. In this section, we highlight the advantages of simulation, after discussing the limitations of the alternative solutions, and then describe the research challenges in large-scale network simulation.

1.1.1 Limitations of Alternative Solutions

Successfully addressing the challenges facing the current Internet is contingent on a deep understanding of its system-wide network behavior. Directly monitoring network state is able to provide an accurate description of the network dynamics, such as the time-variation of traffic intensity, queueing delays, and packet loss rates. However, traffic monitoring at fine time scales on high-speed links itself impose heavy burdens on both processing and storage. In addition, we are still lack of a coordinated measurement framework because of the decentralized, administratively autonomous nature of the Internet. An alternative technique, *network tomography*, is sometimes used for network state estimation in a large-scale network. Statistical methods, including expectation-maximization, likelihood-based analysis and sequential Monte Carlo algorithms, have been applied to infer network attributes such as traffic matrix [138][17], delay [26][27], bandwidth [70], and loss

rates [14]. Although network tomography is a promising tool to characterize the internal network attributes, it has its own limitations. The knowledge by way of statistical inference from end-to-end measurements is still incomplete and prone to inaccuracy; deploying measurement/probe schemes and inference algorithms in Internet-scale networks is difficult, requiring close cooperations among multiple locations; active probing packets used for network tomography may consume excessive communication resources, and they may also be filtered by some firewalls because of policy violations [18].

Any proposed solution to a specific problem in the current Internet should be rigorously and extensively evaluated before its wide deployment. A flawed protocol design or implementation may be vulnerable to malicious attacks, or result in severe performance degradation; techniques that work well for small networks may have poor scalability, and hence fail noticeably when the network size scales by several orders of magnitude. Therefore, the capability of evaluating the performance, security, or dependability of networks with controllable configurations is highly demanded in network design, procurement, and protection.

Theoretical analysis, based on mathematical constructs, is one of the basic techniques used for this purpose. For instance, queueing theory has been widely applied to quantify the performance of computer systems and communication networks [67][72][57]. Although mathematical analysis is a powerful tool to achieve a thorough understanding of the system behavior, a large network like the Internet is often too complex to be within its reach. On the other hand, the mathematics become complicated in solving certain types of problems. In queueing theory, non-Markovian customer arrivals, such as self-similar inputs observed from realistic networks, render analysis of finite buffer systems difficult, and queueing analysis of feedback control systems, exemplified by TCP closed-loop congestion control, is also an arduous undertaking [114]. Mathematical models, in many cases, are simplified or approximated for the sake of tractability, from which inaccuracy can result.

Another evaluation approach is to build real experimental testbeds. The PlanetLab project[3] is such an example that provides an real overlay network platform for developing, deploying, and accessing distributed services, such as content distribution networks, routing overlay networks, and peer-to-peer file sharing [23]. As of December 2004, it has covered more than 500 nodes, which are located in many countries. Experimental testbeds like PlanetLab offer the realism that can hardly be entirely captured by other methods; they are not, however, equivalent to real networks. For instance, the virtualization mechanism adopted by PlanetLab is only a means of achieving scalability and providing security on this computing platform, and resource contentions among processes on the same node may distort the accuracy of the experimental results. In addition, building a large real testbed requires a significant amount of investment on hardware equipments. Finally, experiments done on real testbeds suffer poor repeatability because the computing environment cannot be replicated.

1.1.2 Advantages of Simulation

As alternative solutions are insufficient to fully appreciate the dynamic network behavior or accurately evaluate the new solutions proposed, simulation stands out as an important tool to fulfill these

[3] http://www.planet-lab.org

tasks. Simulation can help us gain deep insights into the Internet's complicated operational characteristics. It not only eliminates the necessity of deploying network measurement or probing tools at a large scale in the Internet, but offers more important details that are often ignored by mathematical analysis. For example, worm scanning traffic can spread over the whole Internet address space. Owing to operational complications, it is difficult to form a global worm traffic monitoring framework that involves all the autonomous systems in the Internet. On the other hand, the classical epidemic model [31], which has been widely applied to study worm propagation [78][98][145], can only conceptually depict the whole process. Some important factors in real networks can hardly be captured: worm traffic may be throttled because of limited communication bandwidths [96]; improved scanning strategies [133] can make it difficult to apply mathematical analysis directly; some effects caused by worm traffic (e.g., BGP routing instability [53]) can impact the propagation process in return, which thus forms a feedback system. Rather, simulations can be easily set up in laboratory settings, and variables that are suspicious of impacting worm propagation can be configured in a controlled way. Bearing these advantages, simulation has been an important tool to investigate worm behavior, and it has been shown able to closely reproduce the effects of work attacks observed from real networks [77][117][145].

Network simulation has played an important role in evaluating new protocol designs during the course of the Internet's evolvement. It enables new theories and techniques to be fully tested under varied network conditions before their wide deployments in the Internet. In laboratory setups, testing scenarios can be generated in a cheap way. For example, the *ns*[4] network simulator has been widely applied in designing new protocols or modifying old protocols. Such protocols include TCP, Internet service models, scheduling or queue management in routers, multimedia, multicast, web caching, wireless sensor networks, satellite networks, and so on[5]. In addition, a lot of research on improving the convergence, security, or scalability of the BGP protocol used the BGP simulator developed by the SSFNet project[6] for evaluation purpose ([115][107][20], to list a few here).

1.1.3 Challenges in Large-Scale Network Simulations

As simulation has established itself as an indispensable tool for Internet-related research, it naturally comes to the question: is simulation of an Internet-scale network feasible? As described at the beginning of this chapter, the current Internet has hundreds of millions of nodes, and its traffic still grows vigorously at a high rate. How much computation, memory, and disk space will be required if we want to simulate it? The calculation has been done in [124]. It conservatively estimates the number of hosts, routers, links, and traffic loads in the Internet. An discrete event packet-oriented network simulator(e.g., *ns* network simulator) is considered. The requirements on memory and disk space are roughly calculated based on measurements from the *ns* network simulator. It is estimated that simulating an Internet-scale network for a single second generates 2.9×10^{11} packet events, and needs $290,000$ seconds to finish if a 1GHz CPU is used; it also requires 2.9×10^{14} bytes of memory, and 1.4×10^{13} bytes of disk space for logging the simulation results. In addition, in order

[4]http://www.isi.edu/nsnam/ns/

[5]Refer to http://www.isi.edu/nsnam/ns/research/ for a representative list of papers that use *ns* simulator.

[6]http://www.ssfnet.org/

to gain more confidence in the results from a simulation, a long simulation run or many independent simulation runs are necessary; this can further prolong the whole execution time. All these lead to a conclusion that packet-oriented simulation of an Internet-scale network is a computationally prohibitive task.

Given the importance of simulation in networking research, it demands more efforts on improving the performance of network simulation. This thesis is motivated by the current challenges in simulating large-scale networks like the Internet. Significant consequences include but are not limited to the following:

- the times spent in waiting for simulation results are reduced dramatically, and less investments on hardware device are required;
- researchers are equipped to explore issues that are indistinct in small networks but manifest themselves in large networks;
- simulations that can be executed in real time can be employed to emulate the real network conditions for protocol testing;
- simulations that satisfy real-time constraints can be used in online cyber-exercises that involve human interactions;
- simulations that can be executed faster than real time can be applied to control and optimize a real operational network in an online manner.

1.2 Research Objectives and Contributions

As illustrated in Section 1.1.3, simulating large-scale network traffic at the level of individual packets is both time- and memory-consuming. With a discrete event packet-oriented simulator, a vast number of simulation events are inevitable in order to model the behavior of the large population of packets in an Internet-scale network. Following such an observation, a question comes naturally:

Question 1 *can the network traffic, or at least part of it, be represented and simulated more efficiently without loss of accuracy in other forms rather than at the level of individual packets?*

Packet-level descriptions depict the traffic behavior directly observed from physical networks, but they are not the only form that is able to capture the network dynamics under analysis. Fluid-based modeling, which represents network traffic as continuous functions of packet rates with time, is another technique to characterize network behavior. Although unable to produce some packet-level details such as jitter, it can still help us study network characteristics like bandwidth consumption [109] or flow throughput [95]. Conceivably, computation costs vary with the traffic representations used in the simulation. From a performance perspective, it is natural to describe network traffic in such a way that the whole computation expense is minimized, as long as the network behavior being investigated can be reproduced as what happen in real networks. This may require that

network traffic be represented at different abstraction levels in the same simulation. The second question, then, is:

Question 2 *can different representations of network traffic be integrated into the same framework without distorting the network characteristics under analysis?*

Conventional wisdom on scaling simulation is on putting more hardware resources for this purpose. It is important to parallelize simulation of network traffic at multiple resolutions on a distributed computation architecture so that more processing power and memory space can be leveraged. In this way, advantages from both fields of high-performance computing and multi-resolution modeling can be combined to improve the performance of large-scale network simulation. It then comes to the third question:

Question 3 *can simulation of network traffic represented at multiple abstraction levels be scaled on a distributed memory multiprocessor?*

This thesis is aimed to address the above three questions, and its primary objective is to investigate efficient techniques for simulating network traffic at multiple resolutions in both sequential and distributed computing environments. The major contributions made in this dissertation are summarized as follows.

- A rate-smoothing technique is developed to mitigate the "ripple effect" in event-driven fluid-oriented simulation. The "ripple effect", if not controlled, can severely undermine the advantages of fluid-oriented traffic models. The rate-smoothing technique effectively prevents the explosion of simulation events by exploiting the "insensitive" period that fluid rate changes have to suffer when traversing a link. This approach can provably dampen the "ripple effect".

- A mechanism is developed to seamlessly integrate multi-resolution network traffic representations into the same event-driven simulation framework. Mutual interactions between traffic represented with packet-based models and fluid-based models are taken into consideration. Empirical results with a fluid-based TCP model show that hybrid simulation can achieve significant execution speedups against the pure packet-oriented simulation.

- A time-stepped fluid-oriented simulation technique is developed to simulate network traffic in large networks at coarse time scales. It aggressively searches for ports at which all input flow rates can be calculated, and then applies fixed point iteration to determine the unknown flow rates. This approach enables real-time coarse-grained traffic simulation of a network with 3 top-tier ISP backbones using an ordinary PC.

- The time-stepped fluid-oriented simulation technique mentioned above is parallelized on a distributed memory processor. Non-committal barriers are used to synchronize participating processors. Problem-specific knowledge is exploited to reduce the synchronization cost. Excellent scalability has been observed in the experiements with both fixed and scaled network sizes.

1.3 Thesis Organization

The remainder of this dissertation is structured as follows. Chapter 2 provides background information for this thesis. In the first part, it introduces the definition of simulation and different simulation models. In the second part, it summarizes existing approaches to improving the performance of network simulation, including model simplification, simulation event management algorithms, parallel and distributed simulation, computation sharing, and variance reduction.

Chapter 3 provides an overview of this dissertation. It describes the multi-resolution traffic simulation framework, and also presents some details on its implementation in the iSSFNet network simulator.

Chapter 4 starts by introducing an implementation of a fluid FIFO multiplexer, and the "ripple effect" is explained thereafter. A rate-smoothing technique that provably mitigates the "ripple effect" is then described. In this chapter, we also describe the algorithms to integrate both fluid- and packet-oriented models into the same framework. This chapter further introduces a modified fluid-based TCP model, and presents experimental results on the performance gain from the hybrid simulation of TCP traffic against pure packet-level simulation.

Chapter 5 describes a time-stepped fluid simulation technique that simulates network traffic at coarse time scales, and then uses empirical experiments to illustrate the convergence property and performance of the algorithm. Afterwards, this chapter discusses how to parallelize the time-stepped fluid-oriented simulation technique on a distributed memory multiprocessor.

Finally, Chapter 6 summarizes the conclusions made in this dissertation and sketches the directions for future research.

Chapter 2

Background

Simulation is an important tool in networking research. As the connectivity of the Internet continues to grow rapidly, simulation helps us understand its operational dynamics and evaluate new theorems and algorithms that are brought forward to improve its performance, security and robustness. This chapter provides background knowledge on what is simulation, what is involved in a simulation study, and how to accelerate simulation, particularly network simulation.

This chapter is structured as follows. Section 2.1 provides the definition of simulation, its life cycle, and different system simulation models. Section 2.2 presents some general approaches to simplifying simulation models and also discusses how to reduce the complexity of traffic models and routing protocols in network simulation. Section 2.3 gives a brief survey on simulation event management algorithms; an optimization in network simulation is also discussed. In Section 2.4, two general approaches to parallelizing simulation are presented; it further presents two families of synchronization protocols in parallel and distributed simulation; thereafter, this section gives a brief introdution to existing parallel network simulators and some load balancing techniques in parallel network simulation. Section 2.5 describes how to share computation in simulation. Section 2.6 summarizes some general approaches to reducing the variance in simulation results. The final section gives a summary on this chapter.

2.1 Simulation

2.1.1 Definitions, Motivations and Procedures

Simulation can be defined in different ways. Two representative definitions are given as follows.

Definition 1 *"A simulation is the imitation of the operation of a real-world process or system over time."* [6]

Definition 2 *"Simulation is the process of designing a model of a real system and conducting experiments with this model for the purpose either of understanding the system or of evaluating various*

strategies (within the limits imposed by a criterion or set of criteria) for the operation of the system." [129]

The first definition interprets the primary task of a simulation as reproducing the evolution of a real system. The second one, rather, emphasizes its motivations and the procedures involved.

Simulation can help us obtain deep insights into the behavior of a real system, especially of one that is too complicated to be mathematically tractable, and evaluate alternative plans in system design. Analytical methods, albeit capable of producing results quickly when they are mathematically well-studied processes, become intractable in studying large, complex systems. Therefore, simplifications or approximations, unavoidably, have to be introduced for the sake of tractability; inaccuracy may result from this process. Simulation also requires simplifications when the simulation model is built. Hence, it is usually not a perfect "clone" of the real system. However, simulation is capable of solving some mathematical models numerically that are analytically intractable. This power, therefore, eliminates the necessity of over-simplifying the models as in many analytical methods.

Simulation is also helpful in prototyping new system designs. Even though the system has not been built and is still in the design phase, simulation can be used to predict possible execution paths and expose potential design flaws. Simulation is also able to foretell the consequences of replacing an existing solution with a new one in an already running real system without interrupting its normal operation.

By Definition 2, a typical simulation study roughly consists of two stages: model development and experiments. In the first stage, a real-world system is formalized into a set of simulation models that not only characterize the relationships inherent in the system, either mathematically, logically, or symbolically, but also are recognizable and executable by computers; these simulation models are executed on computers in the second stage to run experiments and generate results that expose the system characteristics of interest. The detailed steps involved in a simulation study are illustrated in Figure 2.1, adapted from [71]. A simulation project begins by formulating the problem under analysis so that the project objective can be clearly understood. In the next step, information concerning the organization and the activities of the system is gathered and then used to construct a conceptual model, which is a set of assumptions and algorithms that characterize the behavior of the system; at the same time, the data serving as the input parameters to the model are also collected. The conceptual model derived so far can be incomplete, or even wrong; it is necessary to validate the model before building an executable model. It is also necessary to validate the programmed model in order to ensure the correctness of the simulation results. When the programmed model has been validated as an accurate representation of the real system, it can be used to run simulation experiments. While designing the experiments, modelers should make careful decisions on how long a simulation run should be, how many times a simulation run should be replicated, and what simulation results should be collected. Finally, the simulation results are analyzed and the conclusions are documented.

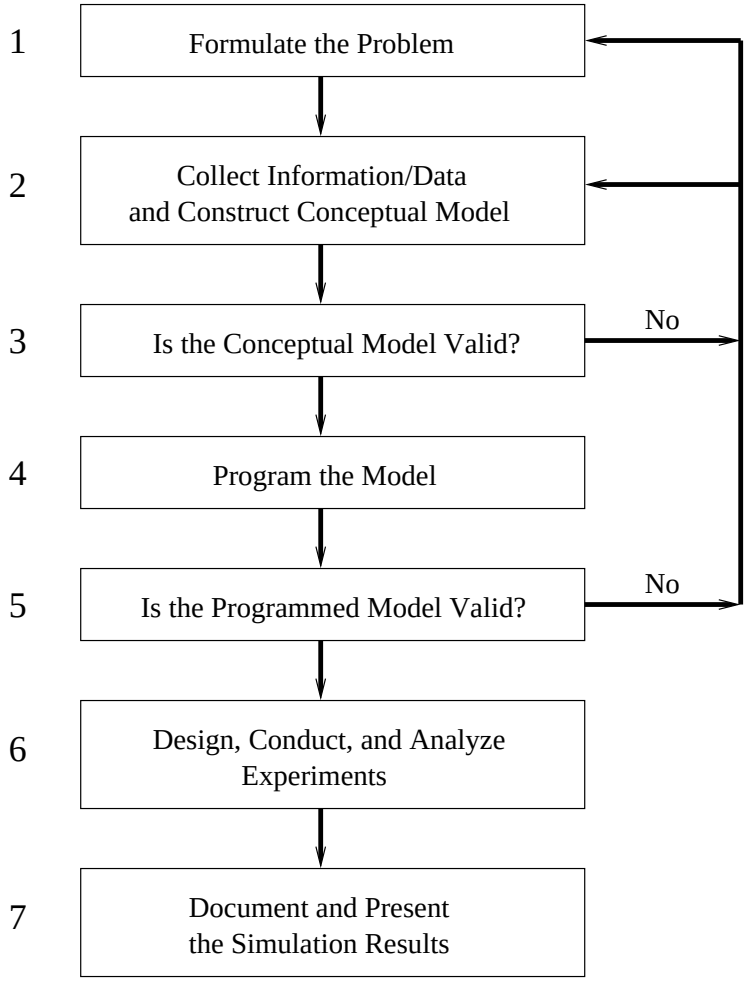


Figure 2.1: The Life Cycle of A Simulation Study

2.1.2 Classifications of Simulation Models

Simulation models can be classified into two broad categories, *continuous models* and *discrete-event models*, based on how the state variables in a model are updated throughout the simulation. In a continuous simulation model, state changes occur continuously as simulation time elapses; by contrast, a discrete-event model changes its states only at discrete time points in simulation. It is important to distinguish means of simulation modeling from the properties of real-world systems. A continuous system like the water in a river can be modeled with both continuous models and discrete-event models; discrete systems such as banks and transportation systems are not restricted to discrete-event simulation models – they can also be characterized by appropriate continuous simulation models. Sometimes, the essential features of a complex system are captured more effectively and efficiently if hybrid simulation models are used. The decision on model selection is contingent on both the properties of the system being modeled and the goal of the simulation study.

Simulation models can also be distinguished by the time management mechanisms used in simulation: *event-driven simulation* (or *discrete-event simulation*), *time-stepped simulation*, and *real time simulation* [6]. In event-driven simulation, simulation events are organized into a list in non-descending order of the fire time of each event. The list is often called *future event list* (FEL). The event at the head of the list (i.e., the one with the earliest fire time) is always processed first. After that, the event is removed from the list, the simulation state is updated, and the simulation time advances to the fire time of the next event. Such a process iterates until the event list becomes empty or the simulation time exceeds the intended simulation length. Event-driven simulation has been applied in simulation of many real systems, such as transportation systems and communication networks, because of its ability to handle asynchrony among objects or entities in a system. In time-stepped simulation, simulation time advances periodically by a constant simulation time unit, which is often called a *time step*. Simulation time moves to the next time step only when all the simulation activities associated with the current time step have been finished. Time-stepped simulation is particularly useful when simulating continuous systems whose dynamics can hardly be characterized with discrete events. The last type of simulation models is driven by wallclock time, a computer's hardware clock time during the execution of a simulation. Real time simulation progresses in synchrony with wallclock time, pacing either in exact real time or in scaled real time; it is mostly suitable for simulation projects in which simulators interact with the real world, such as hardware-in-loop simulation and human-in-loop simulation.

There are some other ways to classify simulations models. For instance, a simulation model can be identified as either static or dynamic based on whether the simulation time advances in the simulation, or either deterministic or stochastic based on whether random processes are used in the simulation.

When simulation modelers develop a discrete-event simulation model, there are three choices of simulation modeling paradigm: *event scheduling*, *process-interaction*, and *activity-scanning* [6]. The event scheduling approach centers on events and how system states change after an event is processed. The event scheduling view conforms to the basic nature of discrete-event simulation and thus easy to understand. However, modelers may find it difficult to abstract the behavior of a complex system into events directly. The process-interaction approach focuses on processes and

their activities, including their interactions with resources. By its root, it is an object-oriented modeling approach; therefore, models developed from this view are well-suited to objected-oriented programming languages. The activity-scanning approach, which concentrates on activities and the preconditions that trigger them, adopts a rule-based mechanism. In this method, simulation time advances at a fixed pace; at each time step, once all the preconditions for an activity are satisfied, it will be executed immediately. Petri Nets models, which are well-studied in Europe, are examples generated from this paradigm.

As simulation has been widely employed to understand the behavior of real systems or evaluate alternative strategies in system designs, it is a nontrivial problem to improve simulation performance, especially when the system being simulated is very complicated. Various techniques exist for this purpose. Based on the methodologies used, they broadly fall into the following categories: *model simplification*, *efficient simulation event management*, *parallel/distributed simulation*, *computation sharing* and *variance reduction*. The following sections give a brief introduction to these techniques and their applications in network simulation.

2.2 Model Simplification

The systems that are investigated with simulation tools grow in both size and complexity. If every detail in the real system under study is captured in the simulation model, the simulation itself can be too computation-intensive with existing computing power. In [112], for example, it is pointed out that the N-body simulation problem is difficult, because the computation workload involved increases more than proportionally with the number of bodies. Complexity of simulation models results from many factors, including both non-technical and technical ones, but the most important one among them is unclear simulation objectives [24]. In many cases, the same simulation objective can be achieved much more efficiently with a simplified model as opposed to a complex one. Hence, given a large, complex real system, it is important to develop a simulation model that contains appropriate level of details so that the computation required is minimized but its validity with respect to the simulation objective is still ensured.

2.2.1 General Techniques

In [38], model simplification techniques, based on which components in the model are modified, are classified into three categories: *boundary modification*, *behavior modification*, and *form modification*.

- **Boundary modification.** This approach aims to reduce the input variable space. It is done by either delimiting the range of a particular input parameter or minimizing the number of input parameters. The latter case conforms to the *parsimonious modeling* principle, which prefers compact models among those that produce equally accurate results. Model sensitivity analysis can be used to identify input variables that hardly affect the simulation results, and these variables can be eliminated from the simulation model.

- **Behavior modification.** In this approach, the states of a simulation model are aggregated, in either space or time domain. At some time point in a simulation, the system state can be decomposed into a vector of state variables. Those variables that are closely correlated with each other in certain ways can be aggregated and then replaced with a single one in the simplified model. This is particularly useful when the dynamics of each state variable before aggregation is of little interest to the modeler and the property of the merged variable after aggregation can be easily defined. State aggregation in space domain is illustrated in Figure 2.2.

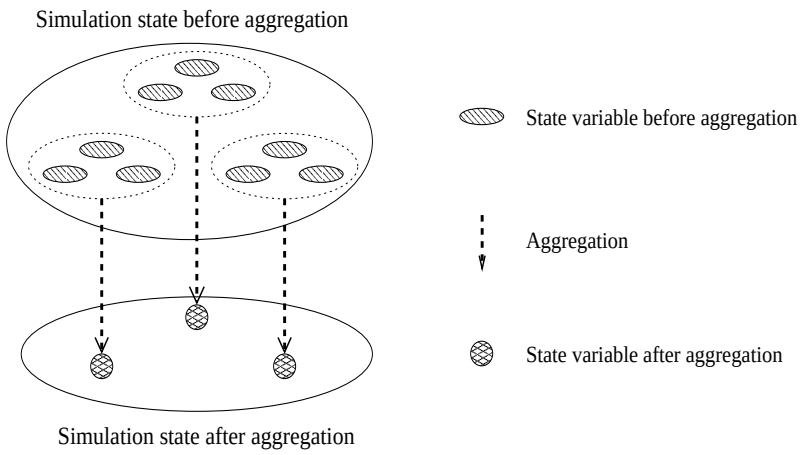


Figure 2.2: State Aggregation in Space Domain

Aggregation in time domain, sometimes, can also reduce the complexity of simulation models. For example, time-stepped simulations of a continuous system vary in their complexity with the time steps chosen to advance the simulation time. Actually, the simulation state at any time step can be seen as the aggregation of all the system states before the simulation time is advanced to the next time step. Therefore, the larger time step, the higher-level abstraction the model provides. On the other hand, temporal aggregation can also happen in event-driven simulation. This is illustrated in Figure 2.3. Discrete simulation events may be aggregated together when their occurrence times are considerably close and they are thus deemed to happen simultaneously.

- **Form modification.** The simulation model in this approach is considered as a "black box", which generates simulation results when inputs are fed into it. In contrast to the previous two approaches, this one replaces the original simulation model or sub-model with a surrogate one that takes a different, but much simpler, form that does the same or approximate input-output transformation. An oft-used technique that adopts this strategy is to generate a lookup table that maps from inputs to outputs directly. If the table size is small, this method provides an efficient substitute for the original model or sub-model; however, as the latter grows in size and complexity, the lookup table can become extremely large. An alternative technique is called *metamodeling* [19]. It seeks a simpler mathematical approximation that statistically

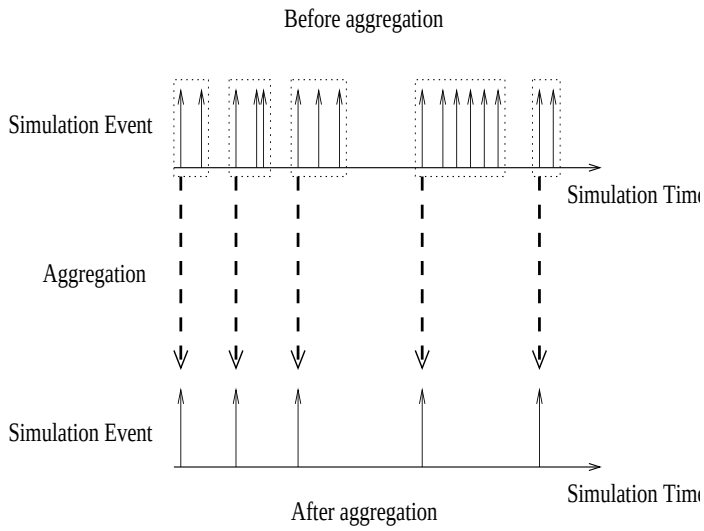


Figure 2.3: State Aggregation in Time Domain

approaches the original model or submodel. Such a mathematical model can be inferred from the input/output data observed in real systems or deduced from the rules that govern the dynamics of real systems. Once such a mathematical model is established, it can be used in simulation to do input-output transformations and generate statistically equivalent results as with the original model.

Model simplification techniques offer the possibility of accelerating simulation of large, complex systems. However, it comes at a price. With details removed from a simulation model, its validity sometimes becomes doubtful. Therefore, it is always necessary to quantify loss of accuracy when a simplified model is adopted, especially in the regions of the input space that are of modeler's interest. On the other hand, in order to minimize computation cost, that real-system objects of the same type are often modeled at different abstraction levels, which is called *multi-resolution modeling*. Seamlessly integrating sub-models represented at multiple abstraction levels in the same simulation model is not always easy to accomplish.

2.2.2 Simplification of Traffic Models in Network Simulation

In a data communication network like the Internet, the packets traversing in it form the network traffic. Simulation capturing high-fidelity details in real networks requires that the behavior of each individual packet be modeled. In packet-oriented network simulators like ns-2, network traffic is represented as discrete packet events and the activity associated with such an event usually involves packet forwarding or application-layer processing. Packet-oriented traffic simulation models, although close to the reality, generate too many events when a large network is simulated. Under this observation, some alternative models have been proposed to reduce the complexity of network traffic simulation models.

The *Flowsim* simulator [1] adopted a packet train model to represent network traffic. A sequence of packets that appear on the same link or reside in the same switch buffer are differentiated into flows according to the conversations or sessions from which they come. A packet train is used to indicate a sequence of closely spaced packets that belong to the same flow. *Flowsim* approximates a packet train with a sequence of evenly spaced packets; a packet train can then be represented with a few fields, including how many packets are carried in this packet train and the occurrence times of the first packet event and the last one. Apparently, if packets are randomly distributed in a packet train, the details on the occurrence time of each particular packet are ignored in this packet train simulation model. The approximation can lead to savings on both execution time and memory, especially when on average many packets are contained in each packet train.

Using event-driven fluid-oriented traffic models to simulate ATM (Asynchronous Transfer Mode) networks was mentioned in [39]. A more thorough discussion on the same topic was given in [61]. Later in [68], similar models were applied to simulate a shared buffer management system. Common to all these models, network traffic is represented as piece-wise constant rate functions of simulation time. Whenever a flow undergoes a rate change, a discrete event that carries the new rate is created to indicate such a change. This is illustrated in Figure 2.4. An on/off traffic source generates three packet bursts, whose rates are R_1, R_2 and R_3 respectively. In total, there are 13 packets. In a packet-oriented simulator, a packet event is generated for each packet when it is emitted from the source; hence, 13 packet events are needed in the simulation. In the event-driven fluid-oriented simulator, the three packet bursts are captured with their respective rates. The packets within each burst are assumed to be constantly spaced. Therefore, event-driven fluid-oriented simulation needs only 6 simulation events, among which 3 events carry the packet rates indicating the heads of the 3 packet bursts and 3 events carry rate 0 indicating their tails. This simple example shows that event-driven fluid-oriented simulation, in some cases, can reduce the number of simulation events in network simulation.

When multiple fluid flows multiplex at the same port, the bandwidth allocated to each fluid flow is regulated by the scheduling policy of that port. As first observed in [61], event-driven fluid-based models, when applied to some scheduling policies, suffer the *ripple effect*. When a change occurs to a flow arrival rate at a congested port that implements, say, a work-conserving FIFO queueing policy, it causes the departure rates of all other flows to change as well. These changes can trigger more rate changes when they are propagated to downstream ports that are also congested. Such a chain of rate changes leads to explosion of simulation events, and thus significantly impair any performance benefit that fluid-based models can offer. Analysis in [79] shows that, under some conditions, packet-oriented simulation even outperforms event-driven fluid-oriented simulation in terms of simulation event rate in a feed-forward FIFO network. We will discuss this phenomenon in more detail in Chapter 4.

Time-driven fluid-oriented traffic models were adopted in [141][15][85]. Like event-driven fluid-oriented models, time-driven fluid-oriented models also represent network traffic as fluid rate functions with simulation time. However, they discretize simulation time into non-overlapping time units. Time-driven fluid-oriented traffic simulation is illustrated in Figure 2.5. At every time step, some fluid-oriented traffic models are sampled to generate fluid rates into the network being simulated. The simulation state of the network at this time step is updated based on both its state at the

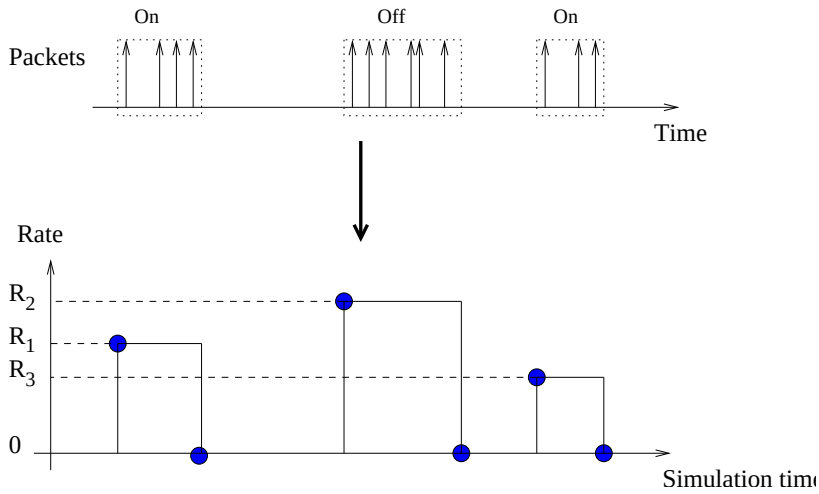


Figure 2.4: Event-Driven Fluid-Oriented Traffic Model from an On/Off Source

previous time step and the newly generated fluid rates. For some closed-loop traffic models like the fluid-based TCP model in [85], the updated network state provides feedback to the fluid-oriented traffic models. Therefore, some of their input parameters will be modified for the next time step. An appealing property of time-driven fluid-oriented models is that the time-driven mechanism naturally provides a sampling process for the fluid-oriented models which are usually continuous functions.

It has been observed that the previous traffic model simplification techniques entirely remove the details of individual packets. This, sometimes, may obscure the simulation's objective. Hence, a hybrid approach called *time-stepped hybrid simulation* (TSHS) technique is proposed in [46]. As in time-driven fluid-oriented simulation, TSHS also discretizes simulation time into small time intervals of equal length. In contrast to time-driven fluid simulation in which only fluid rate information is maintained at each time step, TSHS keeps not only the fluid rate information but also modified packet-level details. Within a time step, packets from the same session within a time step are approximated as evenly spaced so that a fluid rate captures its current traffic pattern. In the meantime, all the packets are linked into a list, which is put in the same data structure with the fluid rate information. The fluid rate information is used for bandwidth competition with other flows in the network as in time-driven fluid-oriented simulation. But when loss happens to this flow, not only the fluid rate should be shrunken, but also some packets in the list be dropped accordingly. The packets, if not dropped in the network, are delivered to the destination. The hybrid mechanism provided by TSHS improves the simulation performance by exploiting abstract fluid-oriented models in the network but is still able to provide packet-level details to the end applications.

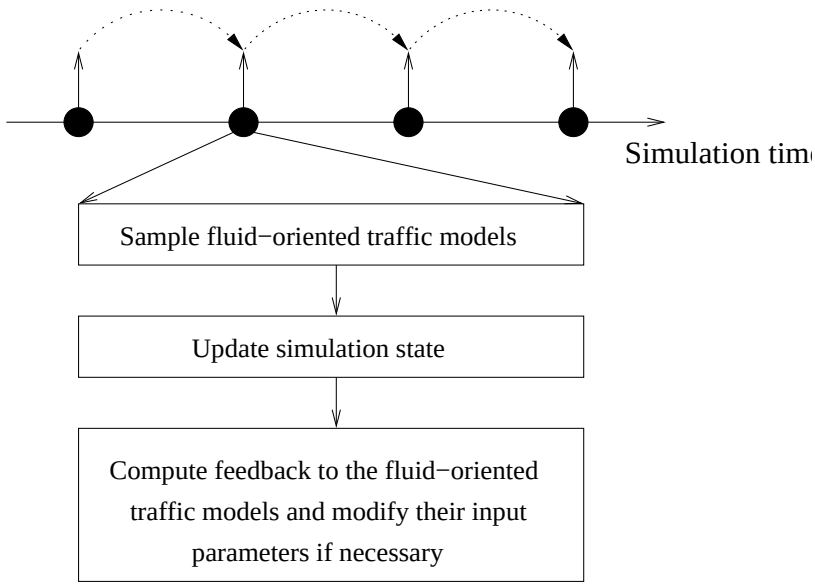


Figure 2.5: Illustration of Time-Driven Fluid-Oriented Simulation

2.2.3 Simplification of Routing Protocols in Network Simulation

The current Internet consists of more than 16,000 autonomous systems (ASes) that are connected with interdomain links. An AS, usually an ISP or a large organization, is a collection of IP networks under a single administrative authority. Typical intradomain routing protocols within an AS include OSPF [100], RIP [47], and IS-IS [16]. These protocols, essentially, compute the shortest path between every pair of nodes in a distributed fashion. On the other hand, BGP [123] is the de-facto interdomain routing protocol used in the current Internet.

Then, how should we model the routing protocols in a network simulator? One approach is to implement them as close as what they are in real networks. Such a strategy is adopted, for example, in the SSFNet simulation package[1]. Apparently, a full-fledged implementation of real routing protocols in a network simulator is helpful in investigating their behavior in dynamic network environments. In many situations, however, this is not the simulation objective. Implementing all the details in a routing protocol is a tedious task and thus prone to errors. Even if all routing protocols can be reproduced in a simulator exactly as in the real networks, inefficiency can result: a lot of computation is spent on route calculation and a lot of memory space is required to store routing tables.

Some simplified routing models were thus proposed. The NIx-Vectors technique [125] eliminates the necessity of maintaining full routing tables on routers. Instead, it computes shortest paths on demand at source nodes and lets each packet carry a vector of routing indices along the path.

[1] http://www.ssfnet.org

At every router, all its Network Interface Cards(NICs) are numbered in order. If a packet carries a vector whose k-th element is i, then when this packet arrives at its k-th hop router, the i-th NIC will be used to forward the packet. Obviously, such an on-demand source routing scheme does not conform to the reality where protocols like OSPF or RIP are used. However, in simulations where routes are static, the NIx-Vectors technique is able to achieve significant savings on both memory usage and execution time.

[49] uses an alternative approach, algorithmic routing, to minimizing memory requirement for routing purpose. This technique maps the network topology under study into a k-ary tree. This process is done by traversing the network with the Breath First Search (BFS) algorithm. For every source-destination pair, there exists only one path in the auxiliary tree structure; this path can be computed with $O(logN)$ time, where N is the number of nodes in the network topology. Path computation is not done on demand, so when a packet arrives at a router, $O(logN)$ time is required to decide to which outgoing NIC this packet should be forwarded. Because the algorithmic routing mechanism does not maintain any routing table, the memory space required in the simulation is reduced. However, as pointed out in the same paper, this approach cannot guarantee that the shortest paths are found between sources and destinations.

Similar to the NIx-Vectors technique, the routing mechanism in [76] also computes routes on demand, and no routing tables are thus necessary. The route is carried with every packet in the simulation. Therefore, at each hop along the path, only $O(1)$ time is needed to identify the outgoing NIC. The primary contribution of this approach is that it exploits the business relationships between ASes to compute interdomain paths. In previous work, it is shown that in order for BGP protocol to converge, interdomain path selection should obey certain rules [42]. These rules are exploited to compute routing paths in network simulations where BGP routes can be assumed to be stable. Empirical results show that such an on-demand policy-based routing scheme is able to achieve significant reduction on execution time and memory usage as opposed to the full BGP implementation.

2.3 Simulation Event Management

As described in Section 2.1.2, discrete event simulation is centered on a future event list in which simulation events are organized based on their fire times. Typical operations on this list include "*schedule-new-event*", "*extract-next-event*", and "*remove-event*". When a new event is generated, the "schedule-new-event" operation is executed and the event is inserted at a proper position in the list; the "extract-next-event" operation extracts the event with the smallest fire time from the list; the "remove-event" operation removes an event with a given identifier from the list. In simulation of a complex real system, the computation cost on manipulating simulation events in the future event list can be significant. Analysis of discrete event simulation of a large-scale computer network shows that, under various conditions, more than 30 percent of the instruction execution counts ascribe to the basic simulation event manipulation operations [29]. Therefore, improving the efficiency of simulation event set algorithms is an important approach to accelerating discrete event simulation.

Data Structures	Schedule-New-Event			Extract-Next-Event		
	Amortized		Max	Amortized		Max
	Expected	Worst		Expected	Worst	
Linked List	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$
Skip List	$O(log(n))$	$O(log(n))$	$O(n)$	$O(1)$	$O(1)$	$O(1)$
Henriksen's	$O(log(n))$	$O(n^{1/2})$	$O(n)$	$O(1)$	$O(1)$	$O(1)$
SPEEDESQ	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(nlog(n))$
Lazy Queue	$O(1)$	$O(n)$	$O(nlog(n))$	$O(1)$	$O(n)$	$O(nlog(n))$
Implicit d-Heap	$O(1)$	$O(log(n))$	$O(log(n))$	$O(log(n))$	$O(log(n))$	$O(log(n))$
Splay Tree	$O(log(n))$	$O(log(n))$	$O(n)$	$O(1)$	$O(1)$	$O(1)$
Skew Heap	$O(log(n))$	$O(log(n))$	$O(n)$	$O(log(n))$	$O(log(n))$	$O(n)$
Calendar Queue	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$

Table 2.1: Analytical Performance of Event Set Algorithms

2.3.1 Simulation Event Set Algorithms

A wide range of data structures have been proposed for managing simulation events in the future event list. Based on the core data structure used, they broadly fall into three categories: *list-based* implementations, *tree-based* implementations, and *calendar-based* implementations. List-based implementations include simple linked lists, skip list [118], Henriksen's data structure [48], SPEEDESQ [134], and lazy queue [122]. *Tree-based* implementations include implicit d-heap, splay tree [132], and skew heap [131]. An example of *calendar-based* implementations is calendar queue [12]. The analytical performance regarding these data structures is depicted in Table 2.1, which is adapted from [121]. It shows expected and worst case amortized cost for both "schedule-new-event" and "extract-next-event" operations; it also gives their maximum costs. The table suggests that no implementation performs dominantly better than all the others. For instance, calendar queue has expected amortized cost $O(1)$ on both "schedule-new-event" and "extract-next-event" operations, but these two operations also suffer worst case cost $O(n)$ when it is necessary to resize the calendar.

Although the analytical performance results presented in Table 2.1 are helpful in understanding the asymptotic behavior of simulation event manipulation algorithms with different data structures, it is also important to examine how they perform empirically when used to simulate real systems. From the experimental results in [121], it is concluded that performance of simulation event manipulation algorithms is application-dependent. More specifically, three observations are made. First, when the average number of simulation events in the future event list is smaller than 1,000, splay tree, skew heap, Henriksen's data structure all behave well in terms of average access times. Second, calendar queue is superior to the other implementations when that number exceeds 5,000. Finally, both splay tree and skew heap show good amortized worst case performance.

2.3.2 An Optimization for Network Simulation

As the Internet evolves, it incorporates more and more high-bandwidth optical links and large-delay satellite and wireless links. A link with high bandwidth-delay product means that many packets can be carried on it simultaneously. When a large network with high bandwidth-delay product links is simulated, putting every packet event into the future event list inevitably leads to a long future event list. From Section 2.3, we know that given any simulation event set algorithm, an increased number of events always mean that more computation is spent on event manipulations in the simulation. An optimization is proposed in [1] to shorten the future event list under such circumstance. It creates a FIFO event queue for each link. At any time, if there are one or more packets on a link, it puts the event representing the packet with the earliest occurrence time on this link into the future event list and all others into the associated FIFO event queue in time order. After a packet event in the future event list is processed, the packet event that is located at the head of the corresponding FIFO event queue is extracted and put into the future event list. This is illustrated in Figure 2.6. Suppose that in a simulation, there are 33 packets that traverse on the link from node A to node B simultaneously. Without the optimization, all these 33 packet events are inserted into the future event list. If the optimization is applied, however, only one packet event appears in the future event list and all the others are temporarily stored in the FIFO event queue. Hence, the optimization significantly shortens the future event list in simulation of high-bandwidth-delay-product networks.

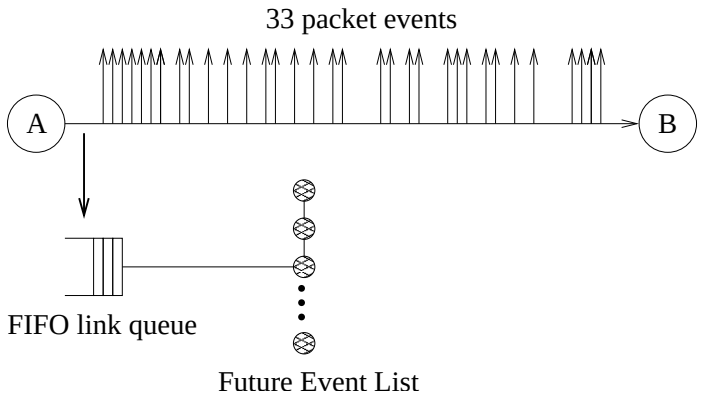


Figure 2.6: Packet Events on A High Bandwidth-Delay Product

2.4 Parallel and Distributed Simulation

Moore's law predicts that the computing power of sequential processors will continue growing rapidly in the near future. However, simulation has been gradually applied to simulate unprecedentedly complex systems such as Internet-scale networks and human brains. Sequential simulation of these systems at fine granularities requires long execution time, even if today's fastest processor is used. Given this, it is natural to accelerate these computation-intensive simulations by leveraging

the computational power of a number of processors. Many real systems can be broken down into individual components which operate in parallel to each other. For example, a computer network is composed of multiple routers that work simultaneously. Such parallelism inherent in real systems can be exploited to improve simulation performance by parallelizing the entire computation workload on multiple processors.

2.4.1 General Techniques

There are two broad approaches to parallelizing a simulation task: *temporal parallelization* and *spatial parallelization*. *Temporal parallelization* divides the time domain of a simulation model into a number of time intervals. Each time interval is simulated on a processor independently. It is likely that the initial states of a time interval mismatch the final states obtained from the previous time interval. Under such circumstance, some fix-up computations are needed to reduce the error. Temporal parallelization was used in [65] to simulate road traffic. However, its success largely depends on whether it is possible to predict accurately the initial states of each time interval; additionally, because simulation models are partitioned in time domain, this method is not suitable for those models that cannot be fit into a single memory space. In *spatial parallelization*, a simulation model is decomposed into a number of components in space domain, and then each component is mapped onto a single processor. The interactions among components in the original model are accomplished with some communication mechanisms like messages. For example, in parallel simulation of a computer network, packets traversing through routers mapped onto different processors can be delivered with communication messages. Spatial parallelization has its own challenges. One of them is that it is not always an easy task to partition the original simulation model in the space domain. In addition, the viability of this approach is contingent on how much parallelism is available in the original simulation model. Despite these challenges, spatial parallelization is a comparatively promising approach to scalable simulation of many real systems, especially when the memory demanded by the original simulation model cannot be accommodated in a single memory space. The following discussions in this section are limited to spatial parallelization.

In the parlance of parallel simulation, a *logical process* corresponds to a sub-model after the original simulation model is partitioned in space domain; interactions between sub-models are represented as event messages between the corresponding logical processes delivered over the underlying communication systems. The architecture of logical process simulation is illustrated in Figure 2.7. The world view it has adopted is the process-interaction paradigm (See Section 2.1.2). With respect to this architecture, the most important problem is: *how do logical processes synchronize with each other?*

There are two families of synchronization protocols, *conservative* and *optimistic*, according to whether logical processes conform to *local causality constraint*: each logical process only processes its events in nondecreasing time orders [41]. Meeting this constraint is sufficient to ensure that simulation results produced be consistent with those obtained from the counterpart sequential simulator. This is the basic foundation of all conservative synchronization protocols; they strictly adhere to this rule so that causality error never occurs. However, nonconformity with this constraint does not necessarily lead to causality errors, because execution of some events within the same logical

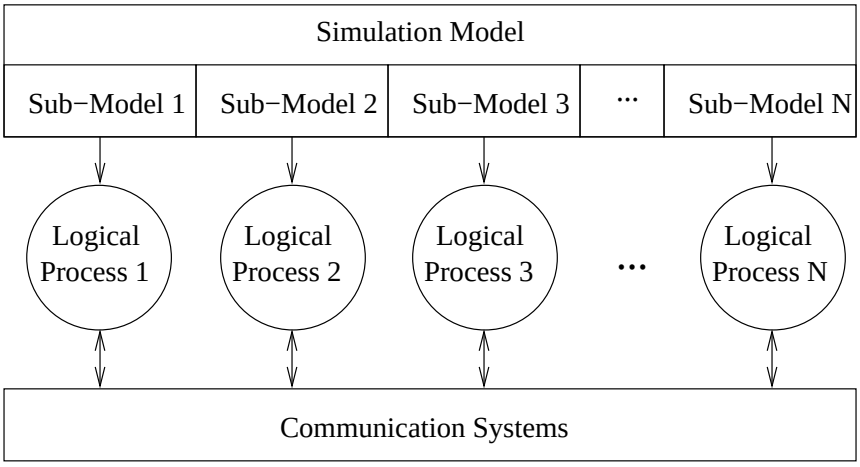


Figure 2.7: Logical Process Simulation

process in different time orders may produce the same results. Optimistic synchronization protocols allow violation of the local causality constraint. Once they detect the occurrence of causality errors, the whole simulation rolls back to a previous safe state.

2.4.1.1 Conservative Synchronization

The Chandy-Misra-Bryant (CMB) synchronization mechanism, among the first several parallel simulation algorithms, was developed in late 1970s based on the conservative principle [21][13]. In this approach, each logical process maintains an incoming link for every other logical process that might send event messages to it. Each link has a FIFO queue that accommodates incoming event messages, and is also associated with a local clock that is set to be the earliest occurrence time among all the event messages in the queue, if it is not empty, or otherwise the occurrence time of the latest event message seen so far. A logical process iteratively picks the first event message from the queue whose local clock time is the smallest; if the queue is empty, the process blocks itself. Straightforward implementation of this mechanism, however, may suffer deadlock. Therefore, some deadlock avoidance or detection techniques were proposed to tackle this problem. For example, *null messages* were used to break the circular dependence among empty queues so that deadlock could be avoided [94].

By its root, CMB algorithm is an asynchronous parallel simulation technique. There is another branch of conservative synchronization mechanisms which use synchronous barrier operations. In these protocols, each logical process iteratively determines a set of events that can be processed safely without violating the local causality constraint. Examples of such protocols include Nicol's conservative protocol [102] and Lubachevsky's bounded-lag protocol proposed [88]. In comparison, asynchronous synchronization algorithms can more fully exploit intrinsic lookaheads, which will be discussed momentarily, but their performance deteriorates when the models involve high connec-

tivity; rather, synchronous mechanisms provide better scalability but they behave poorly when the lookaheads pertaining to the simulation models exhibit high variations. A composite synchronization protocol is proposed in [106] to search an optimal operating point between these two extremes given a specific application.

The success of a conservative synchronization mechanism largely depends on how well it can predict what will, or will not, happen in the future. This capability is also called *lookahead*. The lookahead information can be exploited to determine whether it is safe to process a simulation event. The more such knowledge a logical process can derive, the more simulation events it can possibly process before being blocked to wait for incoming event messages or *null* messages. The lookahead properties are application-specific. For example, an appointment algorithm is proposed in [101] to improve the lookahead ability in simulation of stochastic FCFS queueing networks; the possibilities of lookaheads in wireless network simulation are investigated in [81].

2.4.1.2 Optimistic Synchronization

Time Warp [59][58], developed in the 1980s, was the pioneering optimistic parallel simulation mechanism. As a logical process is allowed to process events without adherence to the local causality constraint, it is likely that later an external event message carrying a time stamp smaller than the local clock arrives. Such a message is called a *strangler message* in Time Warp. The appearance of a strangler message forces the receiving logical process to roll back to a history state earlier than the time stamp carried by this message. In order to recover from causality errors correctly, every logical process in Time Warp has to store its history states and the messages that have been sent and received. On the other hand, some messages may have already been sent to other logical processes before an arriving strangler message invalidates them. Under such circumstance, *anti-messages* need to be sent to the same logical processes in order for them to cancel the erroneous messages received earlier.

As simulation progresses, some history states become unuseful and the storage allocated to them can thus be reclaimed. In order to determine whether a history state recorded at some time point can be discarded, all participating logical processors have to compute collectively the global virtual time (GVT), which is the smallest time stamp among all unprocessed event messages. Given the current GVT, a logical process can safely reclaim the storage allocated for all those history records earlier than GVT. Such a process is called *fossil collection*.

The original Time Warp adopted aggressive cancellation strategy; that is, once a logical process decides to roll back, anti-messages, if necessary, are sent out immediately. However, sometimes the same positive messages (as opposed to anti-messages) are regenerated after the process rolls back. *Lazy cancellation* exploits this possibility and an anti-message is sent only if the previous positive message is not recreated after recomputation. Another optimization technique, called *lazy reevaluation*, applies a similar idea. If the arrival of a strangler message does not alter the state of the process, it is unnecessary to reexecute the events that have been rolled back and the state simply remains as what was before the strangler message was received. There are some other optimization techniques that aim to minimize the storage space used for state saving. For example, incremental or infrequent state saving strategy can reduce the history records that need to be logged [11][86].

In addition, for some specific applications, the state of a process after rollback can be restructured by computing inversely each individual event-processing operation. This technique has been shown able to lessen the computation overhead on state saving in optimistic parallel simulation of some real physical systems [136].

Both conservation and optimistic synchronization mechanisms have their advantages and disadvantages[2]. Compared with optimistic approaches, conservative approaches are usually easier to implement and impose less constraints on memory. However, the performance of conservative synchronization protocols is heavily contingent on lookaheads, whose exploration requires application-domain knowledge; sometimes, it is also sensitive to the changes in simulation models. In addition, owing to the conservative principle, conservative synchronization algorithms cannot exploit the parallelism inherent in simulation models as adequately as optimistic synchronization algorithms. In comparison, optimistic approaches do not rely on explicit lookaheads to achieve good performance, and they aggressively exploit the intrinsic parallelism in simulation models. The flip side of optimistic synchronization protocols includes their relatively high demands on memory space and the complications involved in their implementations.

2.4.2 Parallel and Distributed Network Simulation

In Section 1.1.3, we have described that simulating a large-scale network like the Internet at high fidelity imposes heavy computation burden on a packet-oriented simulator. It is natural to leverage more hardware power and existing parallel and distributed simulation techniques to accelerate this process. The existing parallel network simulators are briefly introduced as follows. The *ns* network simulator has been extended by *pdns*(Parallel/Distributed *ns*)[3] to run in a distributed computation environment. The SSFNet project[4], from its very beginning, aimed to simulate large-scale networks with parallel simulation technology. Its underlying simulation engine, iSSF(previously known as DaSSF)[5], adopts a composite synchronization protocol, which combines the advantages of both synchronous and asynchronous synchronization schemes [106]. The *Telecommunications Description Language* (TED) [116], a language for modeling telecommunication networks, has been used to simulate some large and complex networks, such as ATM Private Network to Network Interface (PNNI) signaling networks and wireless networks; the underlying parallel simulation engine is *Georgia Tech Time-Warp* [40], which is based on an optimistic synchronization scheme. The *Global Mobility Simulation* (Glomosim) [143] aims to simulate an extensive set of wireless networks with a modular design; the underlying simulation engine, *Parsec* [5], can be configured to support both conservative and optimistic parallel simulation. The *TeleSim* framework [137] provides an environment for simulating large-scale telecommunication networks and it works on an optimistic parallel simulation engine. The *USSF* simulator [120], based on the optimistc *WARPED* simulation engine [119], is also able to simulate large-scale network models.

Load balancing is an important problem in parallel simulation, because the simulation perfor-

[2]For more comprehensive discussions on this topic, refer to [41][105][35].
[3]http://www.cc.gatech.edu/computing/compass/pdns/
[4]http://www.ssfnet.org
[5]http://www.crhc.uiuc.edu/ jasonliu/projects/issf/

mance is directly affected by how well the computation load is distributed over all the processors; a poorly designed load balancing algorithm may bring little performance gain from parallel simulation, or lead to even worse performance than sequential simulation because of its extra synchronization cost. In [83], application-specific information is exploited to enhance load balancing in parallel network simulation. The work is based on a conservative synchronization protocol in which processors periodically exchange and process event messages. The time step is so chosen that event messages can be executed without leading to causality errors; in network simulation, its selection is positively correlated with the minimal link latency (MLL) across partitions. One observation is that a larger time step leads to less frequent communication among processors. Hence, a topology-based approach uses static information such as network topology, link bandwidth, and link latency to maximize the MLL across partitions. The extension in [84] adds a threshold on the MLL. The selected MLL across the final partitions are then guaranteed to be larger than this threshold. The second observation is that in order to minimize the communication cost among processors, the amount of traffic that traverses partitions should be minimized. A profile-based approach is thus proposed: traffic profiles are collected from pilot simulation experiments and then used to search an optimal partition plan that minimizes traffic across partitions.

2.5 Computation Sharing

Another approach to improving simulation performance is to share redundant computations. In many situations, multiple simulation runs share the same or similar input data. For example, in sensitivity analysis of a simulation model, a small change is imposed on an input parameter to examine the degree to which it impacts on the simulation outcomes. Evaluation of the sensitivity of a model to an input parameter may require many simulation runs in which the parameter takes sampled values from its feasible region. It is possible that there is only slight difference in the initial settings of these simulations and thus a lot of computations are shared among them. Another example is that in some online simulations, several alternative simulation execution paths diverge from the same decision point. Computations prior to that decision point are exactly the same across all the execution paths.

Based on these observations, some techniques have been proposed to reuse sharable computations across multiple simulation runs. In [44], a *splitting* mechanism is developed to accelerate rare event simulation. The sample path of a simulation is splitted when its execution approaches the occurrence of rare events; therefore, the computations before splitting are shared among the branching execution paths. The *cloning* mechanism in [51] also reuse redundant computations before the branching points at which multiple execution paths diverge. This approach, however, does not physically replicate the whole simulation state at each branching point, which can be expensive in simulation of large, complex systems. Instead, it divides the whole simulation state into finer-grained components called *virtual logical processes*. A clone of the whole simulation state is actually composed of a set of virtual logical processes. Multiple virtual logical processes can be mapped onto the same physical logical processes. Therefore, physical logical processes are shared among different simulation clones. As a simulation clone progresses at its own pace, it may need

to modify the states in some virtual logical processes that are mapped to a common physical logical process shared by other clones. At this time, it replicates the shared physical logical process. The idea is similar to the "copy-on-write" principle in virtual memory management. Another approach, called *updateable simulation* technique [34], exploits application-specific knowledge to improve reusability of shared computations. An initial execution path of the simulation is checkpointed for each event processing. Subsequent simulation runs can reuse those stored states if verification based on application-specific knowledge ensures that no errors are introduced.

In [139], it has been observed that for some applications, computations can also be shared in a single simulation run. For example, in the motivating wireless network simulation, frequent neighborhood calculations can be very computation-expensive. However, mobile nodes, in many cases, move slowly at small time scales and the neighborhood relationships among them thus do not change dramatically. This suggests that the neighborhood computations can be reused in order to improve the simulation performance. A *staged simulation* framework is proposed to exploit computation redundancy in wireless network simulation. In its simplest form, it caches every event, including its function body and a mapping table from the inputs to the corresponding results; therefore, these results can be reused directly when the simulator processes events calling the same event function body with equivalent input data. It is also realized that inputs are often not identical across computations. Therefore, the simulation code is restructured carefully with some techniques so that the redundant computations can be isolated and thus become reusable. In addition, simulation events are sometimes reordered if the final results thereof are not affected but simulation efficiency can be improved.

2.6 Variance Reduction

Simulations involving random components usually produce stochastic outputs. Thanks to law of large numbers, mean of an output process can be estimated by averaging over a number of random simulation trials. Variance, besides mean, is another important statistical measure that evaluates the precision for a series of repetitive experiments. An estimator that produces the same mean but exhibits smaller variance provides better accuracy. An unbiased estimator of variance from n random samples, $x_1, x_2, ..., x_n$, is given as follows:

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2, \tag{2.1}$$

where $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$. Equation (2.1) suggests that the estimator is inversely proportional to the number of independent sample points. A simple approach to reducing variance is, therefore, to increase the number of independent simulation trials. But this is not efficient, especially when a single simulation run takes a long time to finish.

Common variance reduction techniques discussed below aim to improve the quality of outputs without increasing the number of simulation runs.

- **Common random numbers(CRN).** This technique is usually used to compare the performance measures of two (or more) alternative systems. Its basic idea is that both systems are evaluated with the same or similar realization of random components. The rationale behind this is as follows. If X_1 and X_2 are output variates of the two systems, then

$$\text{Var}(X_1 - X_2) = \text{Var}(X_1) + \text{Var}(X_2) - 2\text{Cov}(X_1, X_2), \tag{2.2}$$

  where $\text{Var}(X)$ is the variance of X and $\text{Cov}(X, Y)$ is the covariance of X and Y. With common random numbers, it is expected that stochastic processes X_1 and X_2 are positively correlated (i.e., $\text{Cov}(X_1, X_2) > 0$), and then $\text{Var}(X_1 - X_2)$ can be reduced. Effective implementation of this technique requires synchronization of the random-number usages in both simulation runs; otherwise, increased variances may be observed instead.

- **Antithetic variates(AV).** In this technique, besides output variate X under study, its antithetic variate, denoted as X', is also used. X' has the same mean as X. The variance of their mean, $(X + X')/2$, is

$$\text{Var}[(X + X')/2] = \frac{\text{Var}(X) + \text{Var}(X') + 2\text{Cov}(X, X')}{4}. \tag{2.3}$$

  With this technique, it is expected that $\text{Cov}(X, X') < 0$ so that $\text{Var}[(X + X')/2]$ is reduced. In other words, the negative correlation between X and X' is the precondition to its effectiveness. Applications of this method usually assume that output variate X is a monotonic function of a uniformly distributed variate U. The antithetic variate U' ($U' = 1 - U$) can then be used in the complementary simulation run for X'. Under the AV technique, instead of generating n independent outcomes, $n/2$ pairs of outcomes are generated

$$(X_{2i-1} + X_{2i})/2, \quad i = 1, ..., n/2, \tag{2.4}$$

  where X_{2i} is the antithetic variate of X_{2i-1} computed as above.

- **Control variates(CV).** This approach exploits the positive or negative correlation between the output variate under study and one (or more) control variate to reduce the variance. Suppose the output variate under study is X and the control variate is Y with a known mean $E(Y)$. A controlled estimator, X_c, can be written as $X_c = X - \alpha(Y - E(Y))$. The value of α so picked that $E(X_c)$ is equal to $E(X)$. As for the variance of X_c, we have

$$\text{Var}(X_c) = \text{Var}(X) + \alpha^2\text{Var}(Y) - 2\alpha\text{Cov}(X, Y). \tag{2.5}$$

  The probabilistic rationale of CV is that $\alpha^2\text{Var}(Y) - 2\alpha\text{Cov}(X, Y)$ should be negative and then $\text{Var}(X_c)$ is smaller than $\text{Var}(X)$. Therefore, if X and Y are positively correlated, α should be positive; otherwise if X and Y are negatively correlated, α should be negative. The most important problem with the CV technique is how to choose an appropriate value for α. The control variate Y can be another output variate from the same simulated system, or come from an external system that is similar to, but simpler than, the one being simulated.

- **Conditioning.** Suppose the output variate under study is X and there is another random variable Z. Then,

$$E(X) = E_Z(E(X|Z)) \tag{2.6}$$

and

$$\text{Var}(X) = E_Z(\text{Var}(X|Z)) + \text{Var}(E(X|Z)), \tag{2.7}$$

where $E_Z(.)$ is the expectation of $(.)$ over the space of Z. Because $E_Z(\text{Var}(X|Z))$ in Equation (2.7) must be nonnegative, we have

$$\text{Var}(X) \geq \text{Var}(E(X|Z)). \tag{2.8}$$

Equation (2.8) suggests that if $E(X|Z)$ is known, simulating Z and thus $E(X|Z)$ and then estimating the mean of $E(X|Z)$ can achieve variance reduction as opposed to simulating X directly.

- **Stratified sampling.** The stratified sampling technique partitions the sample space into non-overlapping subgroups or strata, each of which is relatively homogeneous, and draws samples randomly from every stratum. The most important problem with stratified sampling is how to distribute samples over different strata. Common strategies include: (1) *proportionate allocation*, in which the fraction of samples allocated to a particular stratum is proportional to that of its population; (2) *optimum allocation*, in which the fraction of samples allocated to a particular stratum is proportional to the standard deviation of the distribution of the output variate in this region so that the overall sampling variance can be minimized. Proportionate allocation is simpler but may not produce optimal results in terms of variance reduction; under optimum allocation, because the variances of the output variate are not known before simulation, pilot simulations are necessary to gain this knowledge. Stratified sampling can bring significant variance reduction if the subgroups are heterogeneous and appropriate allocation schemes are adopted.

- **Importance sampling.** This approach is particularly useful in rare event simulation, in which events of interest occur only with an small probability. For example, in an ATM network, cell loss rates are typically very small (e.g., smaller than 10^{-9}). In order to observe such rare events, either a large number of simulation runs or a long simulation run is necessary, unless some variance reduction techniques like importance sampling are applied. As its name suggests, the importance sampling technique concentrates on the most important regions in the sample space.

  Suppose we want to estimate $\theta = E_f[h(A)]$ where the PDF (probability density function) function of input variate A is $f(.)$. Another PDF function $g(.)$ satisfies that $g(a) \neq 0$ whenever $f(a) \neq 0$. Then, θ can be rewritten as

$$\theta = E_g[\frac{h(A)f(A)}{g(A)}]. \tag{2.9}$$

In order to estimate θ, n samples of A from PDF $g(.)$ are generated. The importance sampling estimator of θ, denoted as $\widehat{\theta_{is}}$, is

$$\widehat{\theta_{is}} = \sum_{i=1}^{n} \frac{h(A_i)f(A_i)}{ng(A_i)}. \tag{2.10}$$

Note that the original estimator without the importance sampling technique, denoted $\widehat{\theta_o}$, is

$$\widehat{\theta_o} = \sum_{i=1}^{n} \frac{h(A_i)}{n}. \tag{2.11}$$

The selection of sampling distribution $g(.)$ plays a critical role in the importance sampling technique because it decides whether variance reduction (i.e., $Var(\theta_{is}) < Var(\theta_o)$) can be achieved. For example, if $h(a)$ is related to an rare event, $g(.)$ should be so chosen that the regions where $h(a) \neq 0$ are sampled more frequently.

In summary, CRN, AV, CV, and conditioning techniques exploit the already known property of the correlations among the output variates or between the output variates and some auxiliary variates to reduce the variances; stratified sampling and importance sampling techniques, however, restructure inputs into the simulation to achieve the same goal.

2.7 Summary

The first part of this chapter starts by giving two definitions of simulation. The two objectives of a simulation study are summarized as understanding real-life systems and evaluating alternative strategies. The life cycle of a simulation study is also discussed in this part. Because some terms regarding types of simulation models are used throughout this thesis, this part gives a brief introduction to different classifications of simulation models.

The second part of this chapter depicts a big picture on general approaches to accelerating simulation; they include model simplification, efficient event management algorithms, parallel and distributed simulation, computation sharing, and variance reduction. All these techniques play an important role in improving simulation performance. Since the main topic of this thesis is on large-scale network simulation, some application-specific techniques in this domain are also discussed.

Chapter 3

Multi-Resolution Traffic Simulation Framework

The main contribution of this dissertation is to provide a simulation framework that integrates multi-resolution traffic models and also address issues regarding the performance, accuracy, and scalability of these models. This chapter presents an overview of the multi-resolution traffic simulation framework; an example is used to illustrate how to develop traffic models within this architecture. Some implementation details are briefly discussed in this chapter. This chapter is structured as follows. Section 3.1 presents the multi-resolution traffic simulation framework; it also highlights some important problems with respect to it. Section 3.2 briefly discusses some details when we implement this architecture in iSSFNet[1]. The final section summarizes this chapter.

3.1 Multi-Resolution Traffic Simulation Framework

The multi-resolution traffic simulation framework proposed in this dissertation contains traffic models represented at different abstraction levels. Traffic models at high abstraction levels capture the characteristics of network traffic at coarse time scales and ignores details within fine time scales; hence, they are usually less computationally intensive than traffic models at low abstraction levels. On the other hand, traffic models at low abstraction levels provide high-fidelity details of the traffic but may cause relatively high computation workload. Given a particular simulation objective, we should differentiate the network traffic into multiple parts, each of them represented at an appropriate abstraction level according to the simulation objective so that the overall simulation efficiency is optimized.

At the lowest abstraction level, network traffic is modeled at each individual packet, as in a common packet-oriented network simulator like *ns*. Packet-oriented traffic models are necessary when the packet-level details are our interest. For example, some applications like VoIP (Voice Over IP) are sensitive to delay jitters between packets. Simulating traffic generated from such

[1] http://www.crhc.uiuc.edu/ jasonliu/projects/ssfnet/index.html

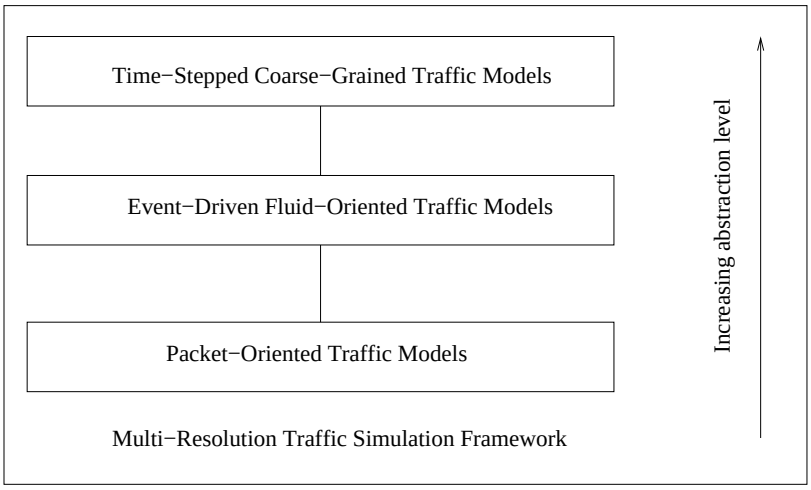


Figure 3.1: Multi-Resolution Traffic Simulation Framework

applications cannot adopt higher abstraction-level traffic models which ignore the behavior of each individual packet. On the other hand, traffic models at this resolution are computationally expensive, because for every packet arrival at a network device, a simulation event needs to be scheduled and processed in the simulation engine.

At the next abstraction level, this framework supports event-driven fluid-oriented traffic models (see Section 2.2.2). Recall that in these models, network traffic is represented as piece-wise constant rate functions of simulation time. A simulation event indicates a rate change for a particular flow. When fluid flows multiplex at the same port, their departure rates are regulated by the scheduling policy adopted by that port. As a fluid rate change arrives at its destination, it is processed by the corresponding application. Event-driven fluid-oriented traffic models are particularly useful for those applications that adapt their sending rates based on feedbacks from the network, such as rate-adaptive multimedia applications. They, however, assume that packets are evenly spaced between rate changes. Such approximations make them unsuitable for situations where packet-level details are of essential importance.

At the highest abstraction level, this framework provides a time-stepped fluid-oriented traffic simulation technique, which models network traffic at coarse time scales. In the space domain, many individual flows are aggregated as long as they correspond to the same source-destination pair; in the time domain, the traffic with regard to the same aggregate flow is further averaged over a large time interval. At this resolution, the simulation time advances in constant time steps. At each time step, each aggregate flow emits a new rate into the network from its source. Because network traffic is modeled at large time scales, what is of our interest is the final state after all aggregate flows converge to a steady state in each time step. The strategy proposed in this dissertation periodically computes the traffic distribution on each link. Compared with event-driven fluid-oriented traffic models, this abstraction level further eliminates the necessity of using discrete events to rep-

resent transient rate changes for individual flows. The time-stepped fluid-oriented traffic simulation technique is particularly useful for background traffic generation in a large-scale network, which is a computation-intensive undertaking if traffic models at lower abstraction levels are used.

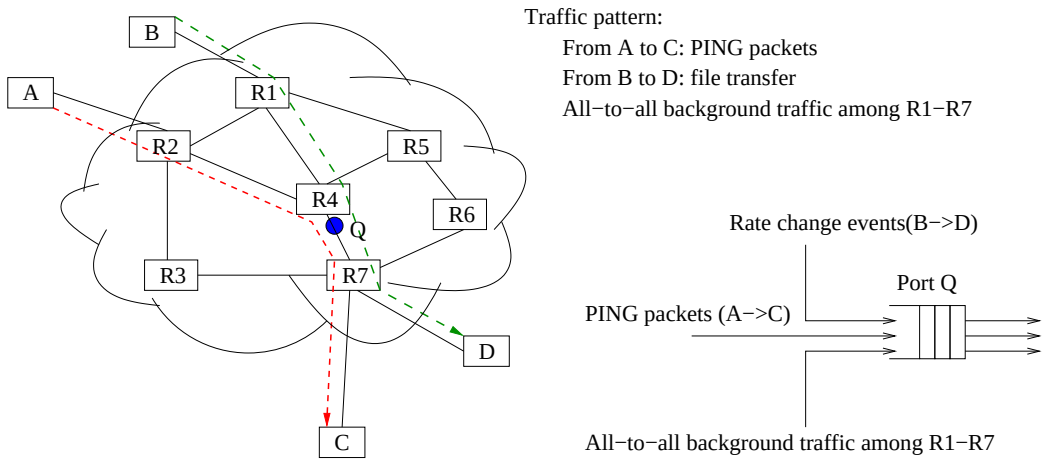


Figure 3.2: An Example Topology with Multi-Resolution Traffic Representations

The example network illustrated in Figure 3.2 is used to explain these multi-resolution traffic models. There are 7 routers, from R1 to R7, and 4 hosts, A, B, C, and D, in the network. The traffic patterns are described as follows. A user at host A randomly sends PING packets to host C. Host B is a file server for a client at host D; when it receives a request from host D, it sends back a file at a constant rate. Host B is not working at a multi-thread mode, so at any time there can be only one ongoing file transfer. Some other traffic is also traversing through the network. The simulation objective is to understand the average delay and loss rate of the PING packets from host A to host C. In the multi-resolution traffic simulation framework, PING packets from host A to host C can be represented with a packet-oriented model. Traffic corresponding to the file transfers from host B to host D can be modeled as an ON/OFF process, in which an ON period corresponds to an ongoing file transfer and an OFF period corresponds to the period when the file server is waiting for another request. An event-driven fluid-oriented traffic model is able to characterize the traffic pattern from host B to host D: when an ON period starts, an event carrying the sending rate is emitted from host B, and when an OFF period starts, an event carrying rate 0 is sent to host D. The remaining traffic, which is distributed on every link in the network, can be very intense compared with the traffic discussed above. Although its behavior is not of interest here, it may impose a significant impact on the PING packets from host A to host D. For example, intense background traffic may lead to severe congestion in the network, causing PING packets to be dropped. This part of the traffic should, therefore, also be taken into consideration. In the framework, we can aggregate the traffic between the same source-destination pair and ignore their dynamics at fine time scales. The time-stepped fluid-oriented traffic simulation technique can then be used here to simulate them as background traffic.

An important problem in the multi-resolution traffic simulation framework is how to integrate multi-resolution traffic models seamlessly. Following the previous example, we can see that in Figure 3.2, both the PING packets from host A to host C and the fluid rate change events from host B to host C will traverse through port Q, which is on router R4 and associated with the link connected to router R7. Part of all-to-all background traffic can also pass this port. Therefore, throughout the simulation port Q will witness traffic at all three different abstraction levels. Ideally, the bandwidth allocation at ports multiplexing multi-resolution traffic should be blind to how the traffic is represented; in other words, it should work as if all traffic is modeled at the finest time scale.

The multi-resolution traffic simulation framework has been parallelized on a distributed memory multiprocessor. High abstraction-level traffic models do not rule out the possibility that the power of parallel computation can be leveraged to improve simulation performance. Instead, the combination of both offers a promising approach to accelerating simulation of large-scale networks. This is particularly important when a simulation model is so complex that it cannot be accommodated in a single memory space. The most difficult part of this aspect in our framework lies on how to parallelize the time-stepped fluid-oriented traffic simulation technique. This is because the traffic load computation at every time step is a global operation and its parallelization thus requires synchronization among processors.

3.2 Implementation in iSSFNet

The multi-resolution traffic simulation framework has been implemented in iSSFNet (the descendent of DaSSFNet). iSSFNet is a network simulator aimed at scalable high-performance network modeling and simulation. It uses the Domain Modeling Language[2] to describe and configure network simulation models. It supports various network protocols, including IP, TCP, UDP, and some physical layer implementations. It also provides several routing schemes, including direct loading of forwarding tables, a static version of OSPF routing protocol, and on-demand policy-based routing protocol [76] (See Section 2.2.3).

The example in Figure 3.3 illustrates how fluid flows are maintained and coexist with other modules in iSSFNet. All fluid-oriented applications are designed on top of the original IP module. Fluid headers are the only means of interaction between IP and fluid-oriented applications. When a fluid-oriented application needs to emit a new rate, a fluid header carrying the new rate is created. Besides the rate, a fluid header also specifies the source port number and the destination port number. When IP layer receives a fluid header from a fluid-oriented application, it treats it as a normal message header like a TCP or UDP header: it simply puts an IP header ahead of it, and after route lookup, pushes it down to the corresponding outgoing NIC. When a NIC receives a regular packet without a fluid header, it processes it as in a common packet-oriented network simulator: compute the queueing delay and schedule a packet event with an occurrence time when this packet departs from the NIC. However, when a NIC receives an IP packet that carries a fluid header, it first forms a session identifier, which is a tuple of protocol identifier, source IP address, source port number,

[2] http://www.ssfnet.org/SSFdocs/dmlReference.html

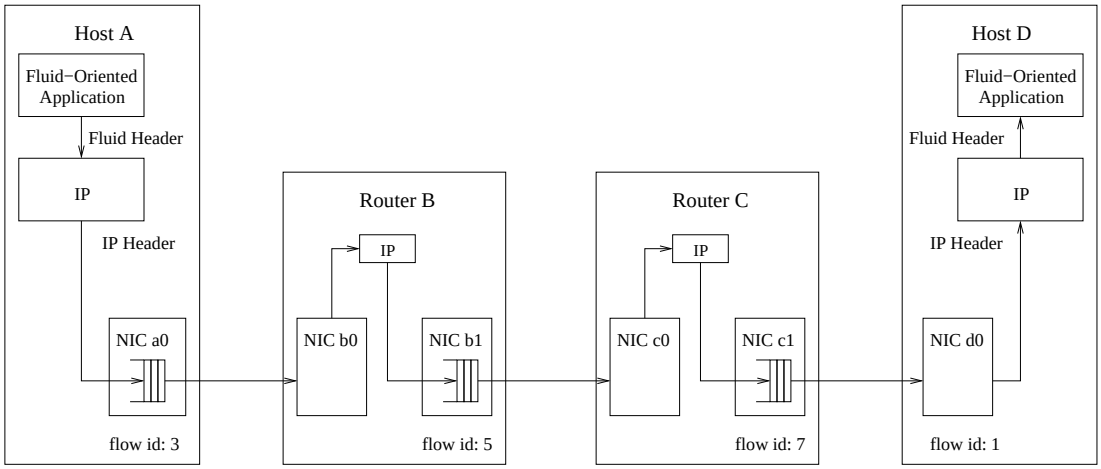


Figure 3.3: Implementation in iSSFnet

destination IP address, and destination port number. It then uses this session identifier to look up a table which maps from session identifiers to local flow identifiers. If the corresponding session identifier cannot be found in the table, a fast flow path setup process is activated: the IP packet is forwarded towards its destination within zero delay in simulation time; every outgoing NIC through which this packet traverses allocates a unique local flow identifier for the corresponding session and also keeps information regarding the next outgoing NIC (if the next hop is the destination, the incoming NIC will be kept instead) and the next local flow identifier; the final incoming NIC also allocates a unique local identifier for this session and keeps the session identifier. For the example in Figure 3.3, the IP packet traverses through outgoing port a_0 at host A, outgoing port b_1 at router B, outgoing port c_1 at router C, and incoming port d_0 at host D along its path. The unique local flow identifiers allocated to the corresponding session are 3, 5, 7, and 1 respectively. These flow identifiers form a static path for this session. Later when host A pushes down new fluid rate changes for this session, these changes are propagated to host D along this static path. As a new rate arrives at NIC d_0, the session identifier corresponding to local flow identifier 1 is used to create an IP packet with a fluid header carrying the new rate. This IP packet is then popped up to the IP layer and then to the receiving application.

This implementation works for both event-driven fluid-oriented traffic simulation and time-stepped coarse-grained traffic simulation. In the interior network, fluid-based computation can use local flow identifiers to locate quickly the next outgoing NIC or the incoming NIC at the destination. This improves simulation performance because it avoids entering IP modules to find an outgoing NIC for every fluid rate change that is propagated across a link.

3.3 Summary

The first part of this chapter has described the multi-resolution traffic simulation framework. In this framework, traffic models at three different abstraction levels are briefly introduced. This part has also discussed how to develop traffic models within this framework. The second part of this chapter uses an example to illustrate how the framework is implemented in an existing network simulator.

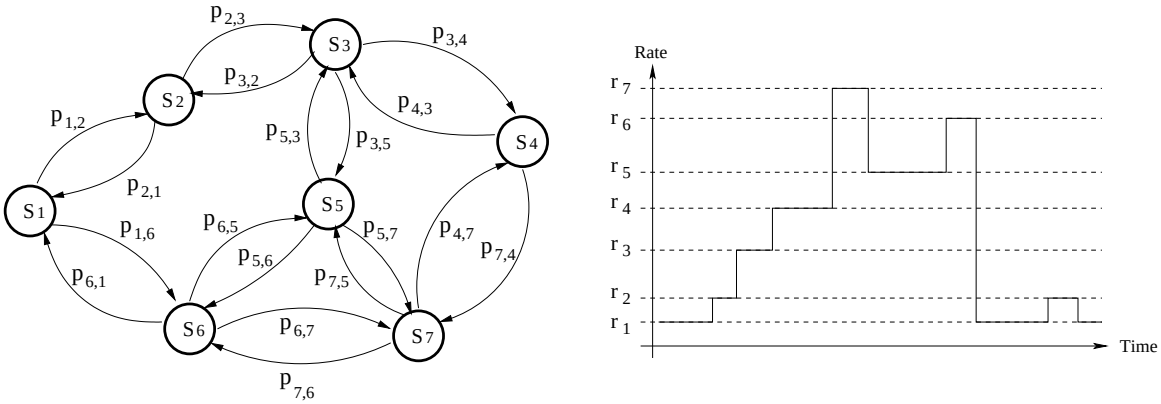
The multi-resolution traffic simulation framework described in the chapter integrates two low-resolution traffic modeling approaches in addition to the conventional packet-oriented modeling technique. The next chapter will delve deeply into the details on event-driven fluid-oriented traffic simulation and its integration with packet-oriented traffic simulation.

Chapter 4

Event-Driven Fluid-Oriented Traffic Simulation

In this chapter we describe event-driven fluid-oriented traffic models in the multi-resolution traffic simulation framework discussed in Chapter 3. In event-driven fluid-oriented traffic simulation, a rate change that occurs to a flow is represented with a discrete event. If such rate changes appear relatively infrequently compared with the number of packets in the simulator, it is expected that event-driven fluid-oriented traffic simulation outperforms the counterpart packet-oriented traffic simulation. However, as events representing rate changes of multiple flows multiplex at congested queues, many new events may be created because of bandwidth sharing. This phenomenon, often called *ripple effect*, offsets any performance gain from using event-driven fluid-oriented traffic models. In this chapter, we present a rate smoothing technique to prevent this from happening. With this method, an upper bound can be established on the number of fluid rate changes received by network components. Another contribution made in this chapter is to seamlessly integrate event-driven fluid-oriented models and conventional packet-oriented models at the same port. Given this integration model, hybrid simulation of TCP traffic is also discussed.

The rest of this chapter is structured as follows. The next section provides background knowledge on event-driven traffic models. Section 4.2 is focused on pure event-driven fluid-oriented traffic simulation. In Section 4.2.1, we introduce an implementation of a fluid multiplexer. The "ripple effect" inherent in event-driven fluid-oriented traffic simulation is discussed in Section 4.2.2. In Section 4.2.3, we describe a rate smoothing technique that provably dampens the ripple effect in event-driven fluid-oriented traffic simulation. In Section 4.3, we present how to integrate mixed traffic representations at the same port. Section 4.4 describes the hybrid simulation of TCP traffic. The related work is introduced in Section 4.5 and the chapter is summarized in Section 4.6.



(a) A Seven-State Markov Chain (b) A Sample Rate Trace Generated

Figure 4.1: Illustration of MMFM Traffic Model

4.1 Background

Fluid-oriented traffic models represent network traffic as a continuous rate function. They are appropriate in cases where the behavior of individual traffic units are of little interest to the modeler. There is a long history of using fluid models to characterize traffic in both telecommunication networks and data communication networks. For example, the Markov Modulated Fluid Model (MMFM) has been applied to model VBR (Variable Bit Rate) video traffic sources [90][128]. The emission rate of a source modeled with MMFM is contingent on its current state in the underlying Markov chain. Associated with each state is a constant rate at which the source emits traffic. In Figure 4.1(a), we give a MMFM model with seven states in the Markov chain. An arrow in the graph represents a state transition with the transition probability shown close to it. Associated with state S_i ($1 \leq i \leq 7$) is constant rate r_i. In Figure 4.1(b), a sample sequence of fluid rate changes generated from this model is presented.

TCP traffic can also be described with fluid-based models. We consider a simple case where TCP protocol is in the *slow start* phase and no loss occurs in the network. In the first round, TCP sender sends out a single data packet at the line speed of the outgoing NIC, denoted by λ_s. In the fluid-oriented TCP model, the rate corresponding to the starting bit of the packet is λ_s, the rate corresponding to the ending bit of the packet is 0, and the time interval between these two rate changes is the transmission time of this data packet. When the packet reaches the TCP receiver, an *acknowledgment* (ACK) packet is sent out. Suppose that the line speed associated with the outgoing interface is λ_r. Similarly, in the fluid-oriented TCP model, the rate corresponding to the starting bit of the ACK packet is λ_r, the rate corresponding to the ending bit of the ACK packet is 0, and the time interval between these two rates is the transmission time of the ACK packet. When the sender receives the ACK packet, it sends out two back-to-back data packets. Hence, in the fluid-oriented TCP model, the rate corresponding to the starting bit of the *first* data packet is λ_s, the

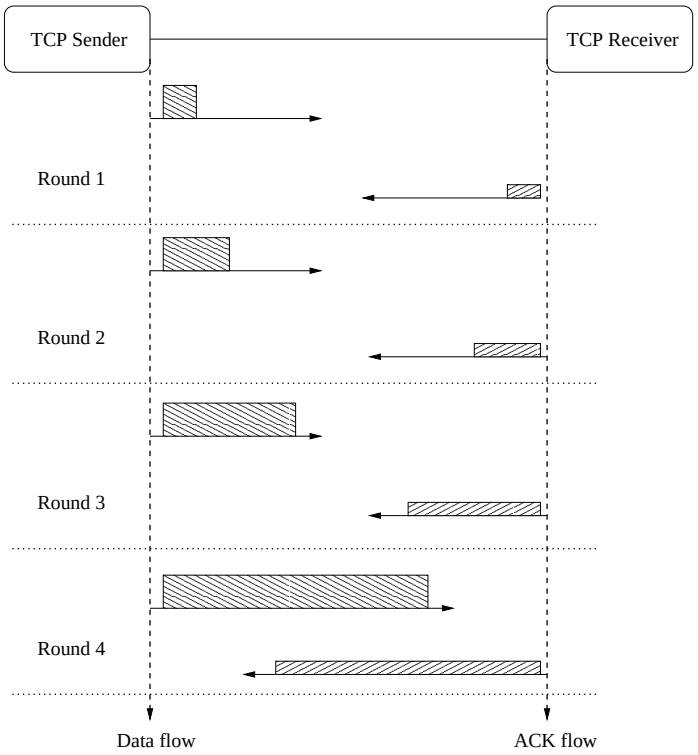


Figure 4.2: Slow Start Phase in A Fluid-Based TCP Model

rate corresponding to the ending bit of the *second* data packet is 0, and the time interval between these two rates is two times the transmission time of a data packet. Suppose that the TCP receiver acknowledges every data packet that arrives, and if back-to-back data packets arrive in a batch, ACK packets are released after all of them arrive. Then, two ACK packets are sent to the TCP sender in the second round. In the fluid-oriented TCP model, the rate corresponding to the starting bit of the *first* ACK packet is λ_s, the rate corresponding to the ending bit of the *second* ACK packet is 0, and the time interval between these two rates is two times the transmission time of an ACK packet. The following rounds are similar to the first two rounds until TCP enters the *congestion avoidance* phase. A much more complicated fluid-based TCP model is introduced in [104]. This model considers both slow start and congestion avoidance phases, maximum TCP window size, lost traffic in the network, fast retransmit, and retransmission timeouts.

Both MMFM and the simple fluid-oriented TCP model described above can be deemed as a traffic source model that injects discrete rate changes into the network. In event-driven fluid-oriented simulation, these rate changes can be represented with discrete events. As the traffic traverses through the interior network, these rate changes are propagated toward their destinations. The question then becomes how to create, schedule, and process fluid rate changes in the network. If there is infinite bandwidth in the network, the problem is trivial – when a discrete event representing a rate change is fired, a new one is created and scheduled to fire at the exact time when the corresponding rate change occurs at the next network device. However, in a network whose finite bandwidth is shared among multiple flows, complications may result from the following two reasons:

- When multiple fluid flows traverse through the same multiplexer, if the bandwidth associated with it cannot serve all the traffic, some traffic needs to be backlogged in the buffer, or even dropped owing to buffer overflow.

- When fluid flows share bandwidth with flows represented with packet-oriented models at the same multiplexer, the bandwidth associated with it should be allocated to fluid flows and packet-oriented flows according to its service discipline.

In the next two sections, we discuss how to address these complications. We particularly focus on the performance aspect of proposed solutions, because under some circumstance, fluid-oriented traffic simulation may be even outperformed by the counterpart packet-oriented simulation [79].

4.2 Fluid Multiplexer

In this section, we discuss how to multiplex multiple fluid flows at the same port. We first give an implementation of a discrete-event fluid port with FIFO scheduling discipline in Section 4.2.1. Then follows a discussion on the ripple effect, a phenomenon inherent in any multiplexer whose bandwidth is shared among multiple fluid flows. We further propose a rate smoothing technique which provably dampens the ripple effect.

Notation	Meaning
$\wedge$	logic and
$\vee$	logic or
$\exists x \in X$ s.t. $p(x)$	there exists x in X that satisfies proposition $p(x)$
$\forall x \in X\{p(x)\}$	for every x in X, it satisfies proposition $p(x)$
$\forall x \in X$ s.t. $p(x)\{a_1(x), ..., a_k(x)\}$	for every x in X, if it satisfies proposition $p(x)$, perform action $a_1(x)$, ..., and $a_k(x)$ in order
$a \leftarrow b$	assign b to a

Table 4.1: Notations

4.2.1 A Discrete Event Fluid FIFO Port

The dynamics of a fluid FIFO port with a finite buffer is described as follows. We assume that packets abstracted in fluid flows are of equal size. We let D denote the set that contains all the flow destinations in the network and Q denote the set that contains all the ports in the network. Consider FIFO port q. Let μ_q denote the link bandwidth associated with this port. We also let d_q be the sum of the propagation delay of the link to which port q is connected and the delay needed to transmit a packet by port q. We call d_q the *insensitive latency* of port q. Let Φ_q be the buffer size in port q. The backlog in the buffer at time t is denoted by $\Delta_q(t)$. We let $F_q(t)$ denote the set of flows that traverse through port q at time t. Note that flows may join and leave the network dynamically and thus the set of flows traversing through a port may change as simulation time elapses. If flow f traverses port q, then its arrival rate into port q and departure rate from port q at time t are denoted by $\lambda_{f,q}^{(in)}(t)$ and $\lambda_{f,q}^{(out)}(t)$ respectively. We define function $\Pi(f, q)$, where $f \in F_q(t)$, as follows:

$$\Pi(f,q) = \begin{cases} q' & \text{if } q' \ (q' \in Q) \text{ is the next output port on flow } f\text{'s path after leaving port } q \\ d & \text{if flow } f \text{ arrives at its destination } d \ (d \in D) \text{ after leaving port } q \end{cases} \tag{4.1}$$

The sum of all arrival rates into port q at time t, $\sum_{f \in F_q(t)} \lambda_{f,q}^{(in)}(t)$, is written as $\Lambda_q^{(in)}(t)$. Variable $\widetilde{t_l}$ is used to record the last time the flow variables were updated. In addition, we let notation t^- be the time immediately before time t and notation t^+ be the time immediately after time t.

We define one type of event and two types of timer: *rate change* event, *buffer overflow* timer, and *buffer empty* timer. Their meanings are suggested by their names: when a rate change event fires, some flow rates are changed; when a buffer overflow timer fires, the backlog in the buffer exceeds the buffer size since that time; when a buffer empty timer fires, the backlog in the buffer becomes 0. Note that timers are actually a special form of discrete event in the implementation. Let $I(e)$ denote the set of flows that change their arrival rates associated with rate change event e. If $f \in I(e)$, let $r_f(e)$ denote the new rate of flow f carried in rate change event e.

In the algorithm setup, $\Delta(0) = 0$ and $\widetilde{t_l} = 0$; at every port, the arrival rate and the departure rate of each flow are initialized to be 0. The source of each flow schedules rate change events according to the traffic model characterizing it.

Case 1	Condition	$\Delta_q(t) = 0 \wedge \Lambda_q^{(in)}(t^+) \leq \mu$
	Action	$\forall f \in I(e)\{\lambda_{f,q}^{(out)}(t^+) \leftarrow \lambda_{f,q}^{(in)}(t^+)\}$
		$\forall x \in D \cup Q$ s.t. $(\exists f \in I(e)$ s.t. $\Pi(f,q) = x)$
		$\{$schedule rate change event e' with fire time $t + d_q$ at x,
		$I(e') \leftarrow \{f \mid f \in I(e) \wedge \Pi(f,q) = x\}$
		$\forall f \in I(e')\{r_f(e') \leftarrow \lambda_{f,q}^{(out)}(t^+)\}\}$
Case 2	Condition	$\Delta_q(t) = 0 \wedge \Lambda_q^{(in)}(t^+) > \mu_q$.
	Action	$\forall f \in F_q(t)\{\lambda_{f,q}^{(out)}(t^+) \leftarrow \lambda_{f,q}^{(in)}(t^+) \times \mu_q / \Lambda_q^{(in)}(t^+)\}$
		schedule buffer overflow timer with fire time
		$t + \Phi_q / (\Lambda_q^{(in)}(t^+) - \mu_q)$
		$\forall x \in D \cup Q$ s.t. $(\exists f \in F_q(t)$ s.t. $\Pi(f,q) = x)$
		$\{$schedule rate change event e' with fire time $t + d_q$ at x,
		$I(e') \leftarrow \{f \mid f \in F_q(t) \wedge \Pi(f,q) = x\}$
		$\forall f \in I(e')\{r_f(e') \leftarrow \lambda_{f,q}^{(out)}(t^+)\}\}$

Table 4.2: Formalization of Cases 1 and 2

Now we discuss how to process a rate change event when it fires. Suppose rate change event e fires at port q at time t. For every flow f in $I(e)$, we update its arrival rate into port q immediately after time t, $\lambda_{f,q}^{(in)}(t^+)$, to be $r_f(e)$. We also update $\Lambda_{f,q}^{(in)}(t^+)$ correspondingly. The backlog in the buffer can then be calculated as follows:

$$\Delta_q(t) = min(\Phi_q, max(0, \Delta_q(\widetilde{t_l}) + (\Lambda_q^{(in)}(\widetilde{t_l}^+) - \mu_q)(t - \widetilde{t_l}))) \tag{4.2}$$

In Equation (4.2), term $(\Lambda_{f,q}^{(in)}(\widetilde{t_l}^+) - \mu_q)(t - \widetilde{t_l})$ is the number of bytes that have arrived into the buffer minus the number of bytes that are released from the buffer since time $\widetilde{t_l}$. In other words, this is the absolute change on the number of bytes in the buffer since time $\widetilde{t_l}$. The backlog in the buffer, apparently, should never exceed the buffer size or be less than 0.

After the backlog in the buffer is updated, we cancel any buffer overflow timer or buffer empty timer if one has been scheduled. Since the aggregate arrival rate into the port has changed, the time that a backlogged buffer needs to drain completely or that an unfilled buffer needs to become overflowed may alter correspondingly. Therefore, if a buffer empty timer or a buffer overflow timer is scheduled before, we need to reschedule it. The computation of flow departure rates is based on the updated backlog in the buffer and the relationship between the aggregate arrival rate into the port and the link bandwidth associate with it. We distinguish four non-overlapping cases, each described as a condition and a set of actions. When the condition to a case is satisfied, the corresponding set of actions are performed, in which new rate change events or timer events may be scheduled. Description of the four cases requires new notations. They are listed in Table 4.1.

We formalize the four cases in Tables 4.2 and 4.3. Case 1 applies when there is no backlog in the buffer and all the input traffic can be served without buffering. The departure rate of each flow is

Case 3	Condition	$\Delta_q(t) > 0 \wedge \Lambda_q^{(in)}(t^+) \leq \mu_q$.
	Action	$\forall f \in F_q(t)\{\lambda_{f,q}^{(out)}((t + \Delta_q(t)/\mu_q)^+) \leftarrow \lambda_{f,q}^{(in)}(t^+) \times \mu_q/\Lambda_q^{(in)}(t^+)\}$ schedule buffer empty timer with fire time $t + \Delta_q(t)/(\mu_q - \Lambda_q^{(in)}(t^+))$ $\forall x \in D \cup Q$ s.t. $(\exists f \in F_q(t)$ s.t. $\Pi(f, q) = x)$ $\{$schedule rate change event e' with fire time $t + d_q + \Delta_q(t)/\mu_q$ at x, $I(e') \leftarrow \{f \mid f \in F_q(t) \wedge \Pi(f, q) = x\}$ $\forall f \in I(e')\{r_f(e') \leftarrow \lambda_{f,q}^{(out)}((t + \Delta_q(t)/\mu_q)^+)\}$
Case 4	Condition	$\Delta_q(t) > 0 \wedge \Lambda_q^{(in)}(t^+) > \mu_q$.
	Action	$\forall f \in F_q(t)\{\lambda_{f,q}^{(out)}((t + \Delta_q(t)/\mu_q)^+) \leftarrow \lambda_{f,q}^{(in)}(t^+) \times \mu_q/\Lambda_q^{(in)}(t^+)\}$ schedule buffer overflow timer with fire time $t + (\Phi_q - \Delta_q(t))/(\Lambda_q^{(in)}(t^+) - \mu_q)$ $\forall x \in D \cup Q$ s.t. $(\exists f \in F_q(t)$ s.t. $\Pi(f, q) = x)$ $\{$schedule rate change event e' with fire time $t + d_q + \Delta_q(t)/\mu_q$ at x, $I(e') \leftarrow \{f \mid f \in F_q(t) \wedge \Pi(f, q) = x\}$ $\forall f \in I(e')\{r_f(e') \leftarrow \lambda_{f,q}^{(out)}((t + \Delta_q(t)/\mu_q)^+)\}$

Table 4.3: Formalization of Cases 3 and 4

then equal to its arrival rate. Since some flows change their arrival rates, these rate changes should be propagated to the next ports on their paths, or their destinations if they disappear into a sink, after the corresponding link latencies. Note that x in the action part can be a port or a destination node.

Case 2 applies when there is no backlog in the buffer and the aggregate arrival rate exceeds the link bandwidth associated with the port. Then, according to the FIFO service discipline, all flows share the bandwidth in proportion to their arrival rates. Similarly, we need to schedule rate change events at the next ports on their paths, or their destinations if they disappear into a sink at the next hop. In addition, since the aggregate arrival rate exceeds the link bandwidth associated with the port, the backlog keeps increasing; therefore, a buffer overflow timer is scheduled at the exact time when the buffer becomes full.

Case 3 applies when there is backlog in the buffer and the aggregate arrival rate does not exceed the link bandwidth associated with the port. Under such circumstance, the backlog in the buffer does not grow. The traffic that arrives after time t but before the buffer becomes empty will be buffered; when it is served, all flows share the bandwidth in proportion to their arrival rates during the above time period. In addition, since the aggregate arrival rate does not exceed the link bandwidth associated with the port, a buffer empty timer is scheduled at the exact time when the buffer becomes empty. Note that if $\Lambda_q^{(in)}(t^+)$ is equal to μ_q, the backlog does not change, so the fire time scheduled for the buffer empty timer is actually in the infinite future. Similarly, we schedule rate change events at the next hops on the paths of the flows. A special case is that $\Lambda_q^{(in)}(t^+)$ is equal to 0. When this occurs, we only need to schedule a buffer empty timer with fire time $t + \Delta_q(t)/(\mu_q - \Lambda_q^{(in)}(t^+))$.

Case 4 applies when there is backlog in the buffer and the aggregate arrival rate exceeds the link bandwidth associated with the port. Under such circumstance, the backlog in the buffer keeps growing. The traffic that arrives after time t will be buffered, or lost after the buffer overflows; all flows share the bandwidth in proportion to their arrival rates during the above time period. In addition, since the aggregate arrival rate exceeds the link bandwidth associated with the port, a buffer overflow timer is scheduled at the exact time when the buffer becomes full. Similarly, we schedule rate change events at the next hops on the paths of the flows.

Now we discuss how to update the state variables when a buffer empty timer fires. Suppose a buffer empty timer fires at time t. We update the departure rate of every flow to be exactly the same as its arrival rate and schedule rate change events at the downstream ports correspondingly. We define the action as follows:

Action: $\forall f \in F_q(t)\{\lambda_{f,q}^{(out)}(t^+) \leftarrow \lambda_{f,q}^{(in)}(t^+)\}$
$\forall x$ s.t. $(\exists f \in F_q(t)$ s.t. $\Pi(f,q) = x)$
$\{$schedule rate change event e' with fire time $t + d_q$ at x,
$I(e') \leftarrow \{f | f \in F_q(t) \wedge \Pi(f,q) = x\}$
$\forall f \in I(e')\{r_f(e') \leftarrow \lambda_{f,q}^{(out)}(t^+)\}\}$

When a buffer overflow timer fires, we do not need to update the flow departure rates. A buffer overflow timer can be used to generate some traffic loss signals for end-host applications that are sensitive to them (e.g., TCP protocol).

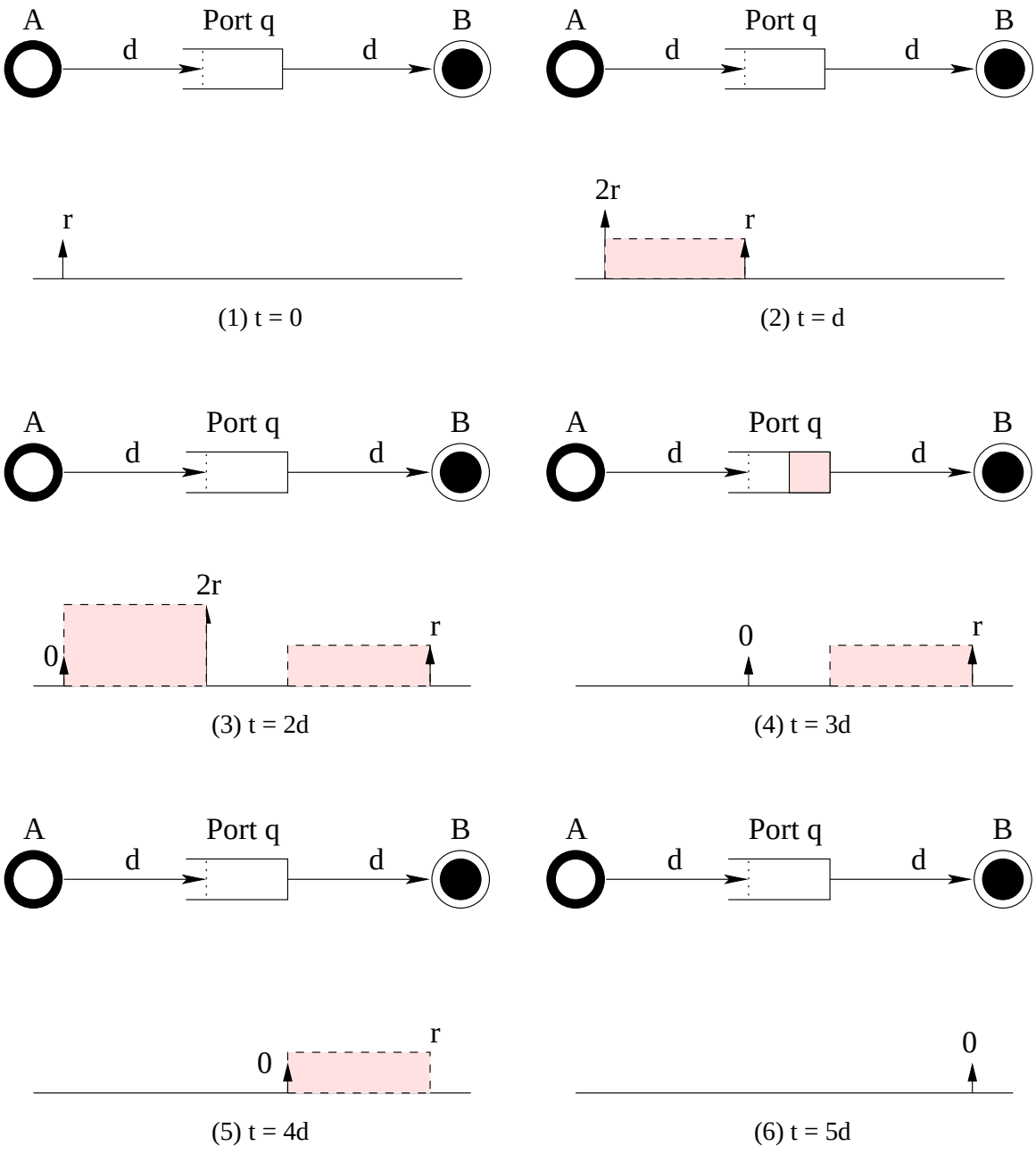


Figure 4.3: Illustration of A Discrete Event Fluid FIFO Port

We use a simple example to explain the dynamics of a discrete event fluid FIFO port. It is illustrated in Figure 4.3. Both traffic source A and sink B are connected to port q with latency d. We ignore the packet transmission time. Source A emits rate r at time 0, rate $2r$ at time d, and rate 0 at time $2d$. The buffer size in port q is $2rd$. At time d, rate r from source A arrives at port q. Case (1) applies: a rate change event carrying rate r is scheduled to fire at time $2d$ at sink B. At time $2d$, rate $2r$ from source A arrives at port q, and Case (2) applies: a rate change event carrying rate r is scheduled to fire at time $3d$ at sink B, and a buffer overflow event is scheduled at time $4d$. At time $3d$, rate 0 from source A arrives at port q. The scheduled buffer overflow timer is canceled and the backlog is updated as rd. Then Case (3) applies: a buffer empty timer is scheduled to fire at time $4d$. Note that it is a special case because the aggregate arrival rate into port q is 0. Hence, no rate change event is scheduled at this time. At time $4d$, the scheduled buffer empty timer at port q fires and a rate change event carrying rate 0 is scheduled to fire at time $5d$ at sink B. This event finally fires at the scheduled time.

4.2.2 Ripple Effect

Consider a rate change event arriving at port q. From Cases (2) and (4) discussed in the previous section, we know that if the new aggregate rate into the port exceeds the link bandwidth associated with it, then for *every* flow traversing through port q, its departure rate must be updated and then propagated to the next hop (after some delay), and for *every* port on the machine at the other endpoint of the link associated with port q, if it is on the path of any flow traversing through port q, there must be a rate change event scheduled at that port. Moreover, if port q is congested, backlog is accumulated in the buffer. From Cases (3) and (4) discussed in the previous section, we also know that whenever a rate change event arriving at a port with backlog, we have to update the departure rate of *every* flow traversing through the port and propagate it to the next hop. When the changes on the departure rates are propagated to the downstream ports, they may also encounter congested ports and trigger new flow rate changes there. In a network where many ports are overloaded, the number of flow rate changes may grow dramatically as simulation time elapses. Such a phenomenon is called "ripple effect" in some literature [61][80].

Ripple effect is inherent in any event-driven fluid-oriented simulation of networks which allow bandwidth sharing among multiple flows. Besides FIFO service discipline, GPS (Generalized Processor Sharing) [113] is another such example. Although GPS scheduling principle provides bandwidth isolation among flows because every flow is provided a guaranteed service rate, it is work-conserving: if a flow does not consume all of its guaranteed service rate, the residual service rate will be shared among other flows whose input traffic cannot be served by their guaranteed service rates. Owing to the possible bandwidth sharing among such flows, event-driven fluid-oriented simulation of a network with GPS service discipline may also suffer the ripple effect.

We use a feed-forward network to illustrate the ripple effect. In this topology, there are 16 flows, from f_1 to f_{16}. Flow f_i ($1 \leq i \leq 4$) traverses 3 ports; flow f_i ($5 \leq i \leq 8$) traverses 2 ports; flow f_i ($9 \leq i \leq 16$) traverses 1 port. Apparently, no circular dependence is formed by these flows in the topology. Such a topology is sometimes called a feed-forward network. Suppose that a change occurs to the arrival rate of flow f_1 into congested port A. Then the arrival rates of flows f_1 and f_2

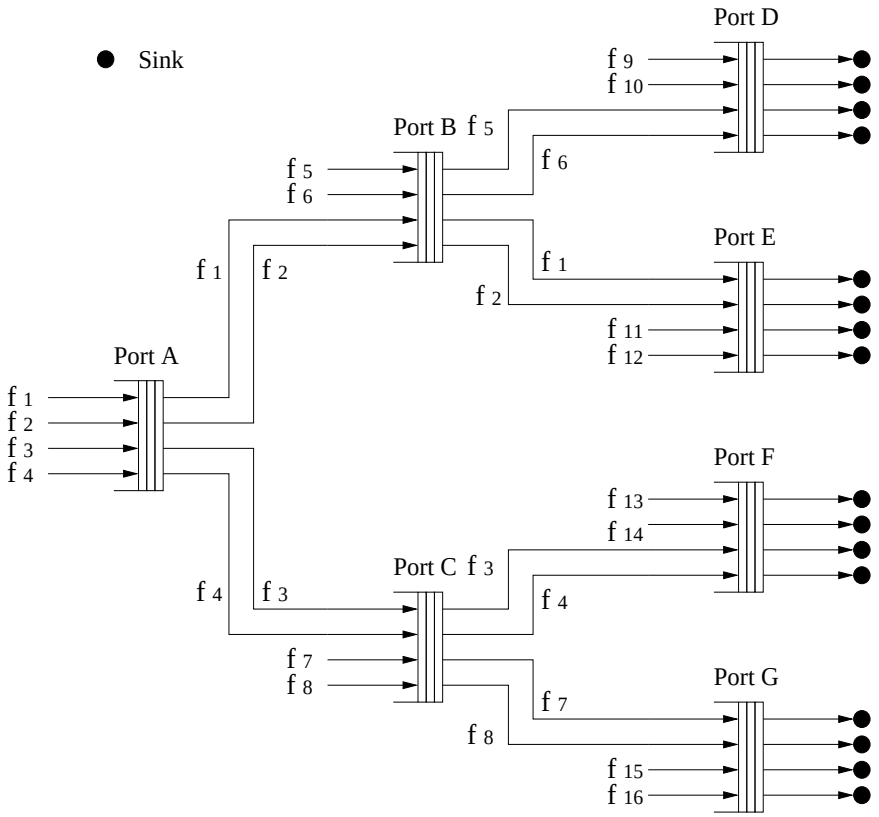


Figure 4.4: A Feed-Forward Topology

into port B and the arrival rates of flows f_3 and f_4 change after some delays. If ports B and C are also congested, these rate changes will be propagated further to ports D, E, F, and G, where flows f_5, f_6, f_7, and f_8 join them as ones that undergo rate changes. Similarly, if ports D, E, F, and G are also congested, there is a rate change event delivered to every flow's destination.

From the perspective of simulation efficiency, ripple effect imposes adverse impact on the event-driven fluid-oriented traffic simulation. As we have seen from the above example, a single rate change at an upstream port may trigger a chain of rate changes downstream in a network where there exist multiple congested ports. Such an explosion of flow rate changes may cause heavy computation cost on processing these rate changes in the simulation.

It is possible that event-driven fluid-oriented traffic simulation that suffers severe ripple effect is outperformed by its counterpart packet-oriented simulation. A thorough comparison between their performance in tandem queueing networks and feedback queueing networks is provided in [79]. Both mathematical analysis and simulation results suggest that when the ripple effect occurs in the network, the rate at which simulation events are generated in event-driven fluid-oriented simulation may exceed that in the counterpart packet-oriented simulation. Recall that the motivation for event-

driven fluid-oriented simulation is to achieve better simulation efficiency by abstracting packet-level details. If it is unable to provide the performance comparable to the packet-oriented simulation, this motivation is undermined.

4.2.3 Rate Smoothing

In this section, we present a rate smoothing solution that provably mitigates the ripple effect in event-driven fluid-oriented traffic simulation. From the four cases when a rate change event is processed at a port, we have observed that once the departure rate of a flow from this port is updated, the new rate is bound to appear at the next hop on the flow's path after a delay. This delay is simply derived as the sum of the insensitive latency of the port and the time needed to release the backlog accumulated before the rate change event arrives. We call this delay an *insensitive period* of a flow rate change.

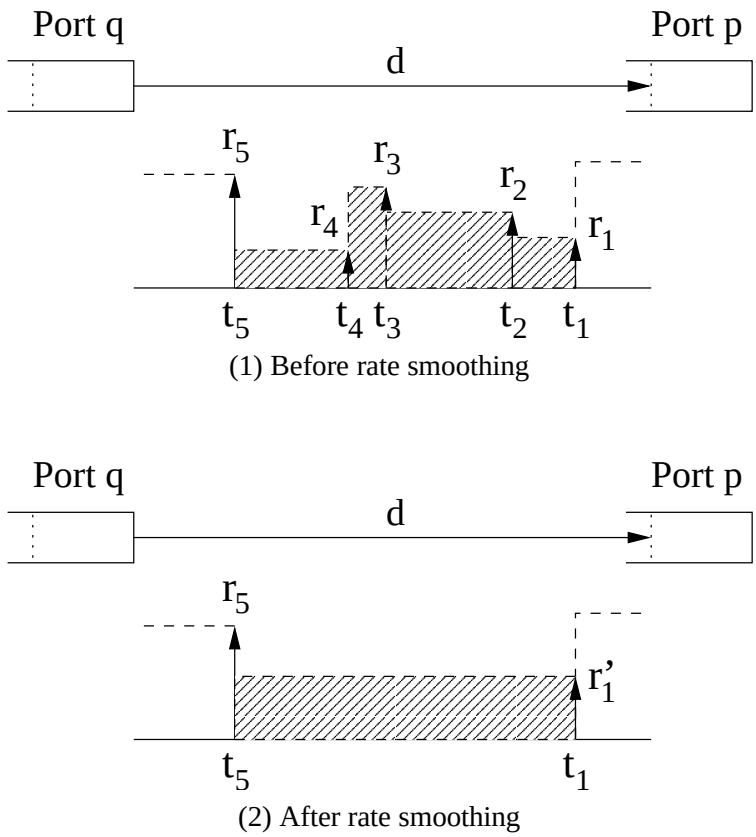


Figure 4.5: Rate Smoothing

It is possible that multiple rate changes of the same flow, after departing from a port, all appear in the insensitive period at some simulation time point. This is illustrated in Figure 4.5(1). There

is a flow that traverses from port q to port p. The insensitive latency from of port q is d. Suppose at some time point, there are 5 flow rate changes that have departed from port q but are still in the insensitive period. These rates are denoted as r_1, r_2, r_3, r_4, and r_5, and their fire times scheduled at port p are t_1, t_2, t_3, t_4, and t_5 respectively. If we flatten the bumpy rates between the flow rate changes whose fire times are t_1 and t_5, as illustrated in Figure 4.5(2), then only 2 rate changes are necessary. That is, the flow rate changes whose fire times are t_2, t_3, and t_4 at port p can be removed from the simulation. The rate change whose fire time is t_1 at port p is replaced with one with rate r_1', where

$$r_1' = r_1(t_2 - t_1) + r_2(t_3 - t_2) + r_3(t_4 - t_3) + r_4(t_5 - t_4). \tag{4.3}$$

Note that the total volume of traffic that arrives at port p does not alter at simulation time t_5 after the rates are smoothed. However, the number of flow rate changes is reduced by three.

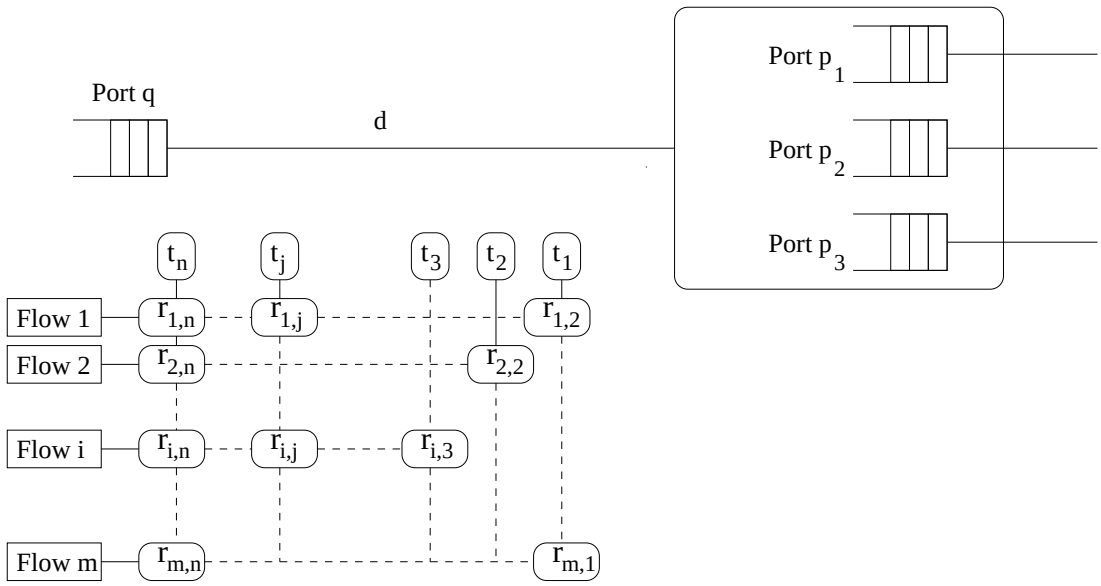


Figure 4.6: Rate Change Matrix

In order to incorporate this idea, we need to modify the implementation in Section 4.2.1. At every port, we maintain a list of pending rate changes for each flow that traverses it. On the other hand, rate changes that occur at the same time are concatenated across flows. Therefore, all the pending rate changes form a matrix-like structure as depicted in Figure 4.6. When processing a rate change event that updates the departure rates of some flows, we create a new column with the appropriate fire time, and concatenate all the new departure rate changes together. When we add a new departure rate change for a flow, we check whether there are more than one flow rate changes whose fire times are no larger than the new one's in the matrix. If there are, we smooth the rates as discussed above and remove the rate changes that happen between the earliest one and the new one. In Figure 4.6, suppose that the column corresponding to fire time t_n is newly added. Then we can

apply the rate smooth technique on flow 1 and remove $r_{1,j}$ from the matrix; similarly for flow i, we can smooth the rate between fire times t_3 and t_n, and thus remove $r_{i,j}$ from the matrix.

We introduce a *rate change delivery* timer, which recursively fires at the earliest occurence time among all rate changes in the matrix. When a rate change delivery timer fires, we process it as follows. We put all the rate changes of flows that go to the same next port into a rate change event, and deliver that event to the corresponding port; that port processes this event instantaneously in the same way as discussed in Section 4.2.1. If there are flow rate changes destined for the node at the other end of the link, we create a rate change event containing all these rate changes and deliver it to the destination node.

From the above description, we can easily obtain the following lemma:

Lemma 1 *Consider any flow traversing through a port with an insensitive latency d. If the rate smoothing technique is applied as described, there are at most 2 departure rate changes within any period of d units of simulation time.*

Proof. We prove it by contradiction. If there are more than 2 departure rate changes within a period d units of simulation time, these rate changes must appear in the rate change matrix simultaneously at some simulation time point. Based on the above description, they can be smoothed and only two are left. This contradicts the assumption. □

With aid of Lemma 1, we can establish the upper bound on the total number of flow rate changes that are received by network components in an event-driven fluid-oriented simulation. We use Q_f to denote the set that contains all the output ports along flow f's path. Recall that d_q is the insensitive latency of port q.

Theorem 1 *Consider an event-driven fluid-oriented traffic simulation. Let set F contain all the fluid flows in it, and let T be the simulation duration time. Then the number of rate changes observed by network components in the simulation is no greater than*

$$\sum_{f \in F} \sum_{q \in Q_f} \lceil \frac{2T}{d_q} \rceil, \tag{4.4}$$

where $\lceil x \rceil$ is the smallest integer that is no less than x.

Proof: For any flow f in the network, the number of its departure rate changes from port q ($q \in Q_f$) that is observed at the next network component after port q is at most $\lceil 2T/d_q \rceil$ based on Lemma 1. Then the number of departure rate changes observed by all the network components on flow f's path is at most $\sum_{q \in Q_f} \lceil 2T/d_q \rceil$. By aggregating over all the flows, we can derive the upper bound on the total number of rate changes observed by network components in the simulation as Formula (4.7). □

From Formula (4.7), we know that the upper bound is entirely dependent on the simulation duration time, the network topology and how flows are routed in the network. Therefore, with the rate smoothing technique applied, it is ensured that uncontrolled exponential explosion of flow rate changes cannot happen.

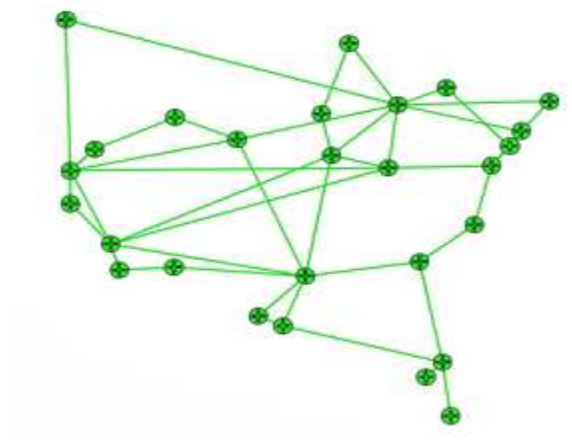


Figure 4.7: The POP-Level ATT Backbone

We use the ATT backbone[1], which is illustrated in Figure 4.7, to empirically study the performance and accuracy of the rate smoothing technique. The topology has 27 POPs (Point of Presences), connected with 100Mbps links. In order to study the impact that link latencies have upon the performance and accuracy of the rate smooth technique, we vary the link latencies in the experiments between 0.1 second, 1 second and 10 seconds. The buffer size in an output port is the product of the bandwidth and the propagation delay of the link associated with the port. For every pair of POPs, we use the MMFM model discussed in Section 4.1 to generate its ingress traffic. Therefore, there are 702 (27×26) fluid flows in total. Each source traffic model is modulated by a Markov chain with two states: in the "on" state, the sending rate is a positive constant r, and in the "off" state, the sending rate is 0. The holding time in each state is exponentially distributed with the same mean that is 1 second. In the experiments, we adjust constant rate r in the "on" state to achieve two levels of traffic intensity in the network, one having average link utilization 50% and the other having average link utilization 80%.

From the experiments, we collect how many flow rate changes have been received by network components. Note that a network component can be either a router or a flow sink. Figure 4.9(a) presents the relative reduction on the flow rate changes observed by network components when the unconstrained rate smoothing technique is applied as opposed to when it is not. It is clear that when the link latency increases, the number of flow rate changes received by network components decreases. This is obvious because a larger link delay leads to more flow rate changes that are smoothed before they take effect at the receiving network components. Moreover, as the traffic intensity level in the network increases, the number of flow rate changes in the simulation also increases because of congestion, causing the number of flow rate changes that are smoothed out to increase. This explains the large gap between the number of flow rate changes under different traffic loads in Figure 4.9(a).

[1]http://www.ssfnet.org/Exchange/gallery/usa/index.html

The number of flow rate changes that are generated in an event-driven fluid-oriened traffic simulation is an important factor in determining its performance. However, there are some other important factors that cannot be ignored. In our implementation, for example, the performance is also affected by the number of timer events fired in the simulation. A timer event is inserted into the future event list in the underlying simulation engine when a rate change delivery timer, a buffer overflow timer, or a buffer empty timer is scheduled. In addition, a traffic source also needs to schedule simulation events to indicate its rate changes. All these suggest that the factors affecting the overall performance of event-driven fluid-oriented traffic simulation are manifold. In Figure 4.9(a), we also present the relative reduction on the execution time when the unconstrained rate smoothing technique is applied as opposed to when it is not. The results tell us that the overall trend of the simulation execution time agrees well with that of the number of fluid rate changes observed by network components in the simulation, but under both traffic loads, the reduction on the total simulation execution time is less impressive than that on the number of fluid rate changes observed by network components.

The performance improvement from the rate smoothing technique has a cost. In order to study how the rate smoothing technique affects the accuracy of the simulation results, for every flow in the topology, we collect the volume of traffic it has received within non-overlapping time intervals of equal length τ. In the following experiments, we set τ to be 1 second. Let $Y_f(k)$ or $Y_f^s(k)$ ($k = 1, 2, ...,$) denote the processes that characterizes the volume of traffic received by flow f's destination within the k-th time interval. Then the average relative error on the volume of traffic received by flow f is defined as

$$E_f = \frac{1}{K} \sum_{k=1}^{K} \frac{|Y_f(k) - Y_f^s(k)|}{Y_f(k)}, \tag{4.5}$$

where K is the total number of non-overlapping timer intervals. In the experiments mentioned later in this section, K is 3,600.

The overall measure of the relative error is obtained by averaging the above relative errors of all the flows:

$$E = \sum_f E_f = \frac{1}{KN_f} \sum_f \sum_{k=1}^{K} \frac{|Y_f(k) - Y_f^s(k)|}{Y_f(k)}, \tag{4.6}$$

where N_f is the total number of fluid flows. In order to reduce the variance in this accuracy measure, we adopt the common random number technique discussed in Section 2.6: when computing a sample of E, the same random numbers are used in the simulations that derive $Y_f(k)$ and $Y_f^s(k)$. In addition, we remove all the samples that are extremely small; otherwise, the unreasonably large relative error of a single sample can cause a very high average relative error.

The experimental results are presented in Figure 4.9(b). It is evident that with larger link delays, more inaccuracy is introduced. For example, when the link delay is as large as 10 seconds, the relative error measured by Formula 4.6 is higher than 300% mark. This suggests that the rate smoothing technique, if used without any constraint, may adversely affects accuracy, especially when the link delays are relatively large. On the other hand, as we mentioned above, a higher traffic intensity

level in the network makes it more possible that flow rate changes are smoothed in the simulation. Therefore, we observe the relative error measure corresponding to average link utilization 80% is higher than that corresponding to average link utilization 50%. However, compared with link delays, the traffic intensity level imposes relatively smaller impact on the relative error measure. This is because a heavier traffic intensity level in the network also leads to more traffic received at the egress nodes within each time interval, thus reducing the relative error measure.

From the experimental results, we have realized that there exists a tradeoff between performance and accuracy when deciding whether the rate smoothing technique should be used. In particular, when the link delays in the network are large, applying the rate smoothing technique without any constraint leads to high inaccuracy. This may not be acceptable under some circumstance. In order to avoid this problem, we can put a time constraint when smoothing flow rate changes. Let Φ be the longest time interval within which flow rate changes are allowed to be smoothed. In other words, we do not smooth rate changes at time scales beyond Φ. Selection of Φ reflects a tradeoff between simulation accuracy and efficiency. A larger Φ can bring better performance but may impair the simulation accuracy to a larger degree.

With the constrained rate smoothing technique, the upper bound on the total number of flow rate changes received by network components in Formula 4.7 should be modified correspondingly. Consider port q of insensitive latency d_q and any flow f that traverses it. If the constrained rate smoothing technique is applied, at most two rate changes are received at the next network component with any time period $min(\Phi, d_q)$. Therefore, we can establish the upper bound on the total number of rate changes in the simulation as in Theorem 1.

Theorem 2 *Consider an event-driven fluid-oriented simulation in which the constrained rate smooth technique is applied as described. Let set F contains all the fluid flows in it, and let T be the simulation duration time. Then the number of rate changes observed by network components in the simulation is no greater than*

$$\sum_{f \in F} \sum_{q \in Q_f} \lceil \frac{2T}{min(\Phi, d_q)} \rceil, \tag{4.7}$$

where $\lceil x \rceil$ is the smallest integer that is no less than x.

We redo the previous experiments by setting Φ to be 1 second. The simulation results are presented in Figure 4.9. As before, the relative reduction on the simulation execution time when the constrained rate smoothing technique is applied is less impressive than that when it is not. This still suggests that the number of fluid rate changes in the simulation is an important but not the only factor that affects the overall simulation performance. Comparing the results obtained by unconstrained and constrained rate smoothing techniques, we have made the following observations:

- When the link delays are 0.1 second, which is much less than time constraint Φ on rate smoothing, the simulation results are hardly affected by whether rate smoothing is constrained;
- When the link delays are 1 second, if rate smoothing is constrained, the relative error decreases but the number of flow rate changes increases slightly. Recall that the insensitive

latency of a port includes both the link propagation delay and the transmission delay of a single packet. Hence, it is actually longer than 1 second. When no time constraint is imposed, the probability with which fluid rates are smoothed is higher than that when time constraint is imposed.

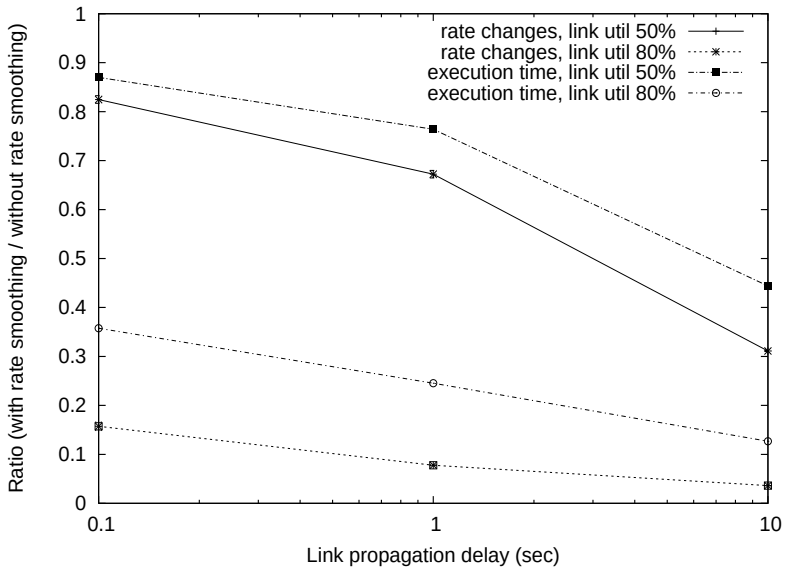
- When the link delays are 10 seconds, if the constrained rate smoothing technique is applied, the relative error drops significantly, but fewer flow rate changes are removed because of the time constraint on rate smoothing. We also observe that as the insensitive latencies of ports exceed time constraint Φ, both the relative error due to rate smoothing and the reduction on flow rate changes remain relatively insensitive to the link propagation delays.

We conclude that the constrained rate smoothing technique, with a careful selection of time constraint Φ, provides flexibility in balancing accuracy and efficiency of event-driven fluid-oriented traffic simulation. The decision on time constraint Φ is application-specific. One important factor that should be taken into consideration is how end applications respond to the inaccuracy due to rate smoothing in the network.

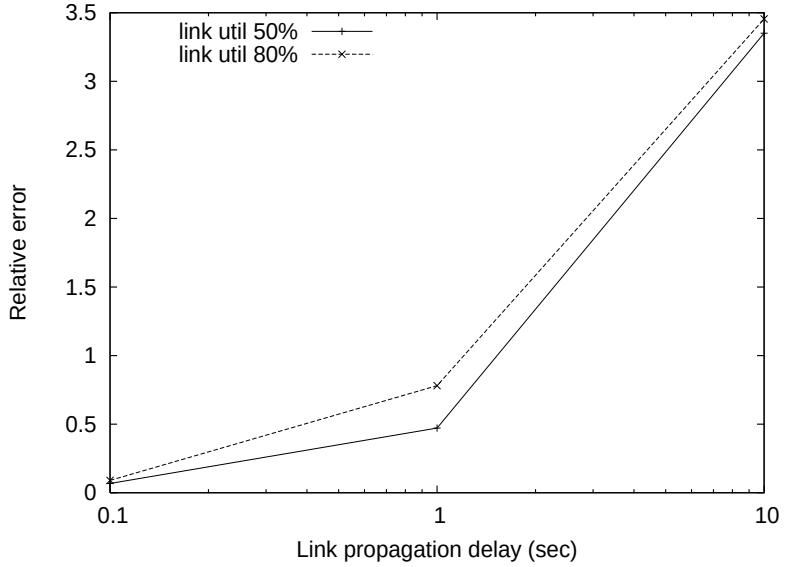
The key idea of rate smoothing technique is to exploit the insensitive latencies that flow rate changes have to suffer when traversing through a link; if they are in the insensitive period simultaneously, we can flatten the rates so that some rate changes can be removed from the simulation. However, there are some complications when the link under consideration crosses timeline boundaries in parallel and distributed simulation (see Section 2.4.1).

- If optimistic synchronization protocol is used, the rate smoothing technique can work as usual. When a rate change delivery timer fires, we pack the rate change events created in the corresponding action part into messages, carrying a proper firing time, and send them to the timelines that contain the receiving network components. When a message carrying flow rate changes arrives at the receiving timeline, if the time these flow rate changes are supposed to fire occurs earlier than the local clock time, the message is treated as a “strangler” event and the receiving timeline rolls back to the time stamp that the message carries.

- If conservative synchronization protocol is used, the rate smoothing technique may not work. Note that in the rate smoothing technique, rate changes are held at the port from which they depart until they take effect at the receiving network components. However, typical conservative synchronization protocols in network simulation exploit the link propagation delays as lookaheads and use them to predict whether it can be ensured that no events will arrive within these delays in the future. The rate smoothing technique violates the assumption on such lookahead, and therefore, potentially causes causality errors. Hence, in parallel or distributed simulation based on conservative synchronization principle, we deactivate the rate smoothing technique on every port that is associated with a link crossing timeline boundaries.

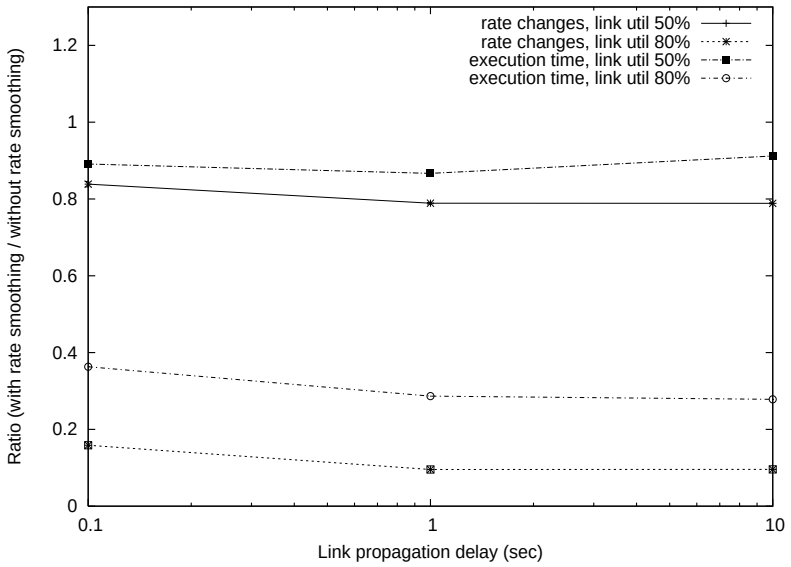


(a) Ratio of flow rate changes and execution time (simulation with unconstrained rate smoothing / simulation without rate smoothing)

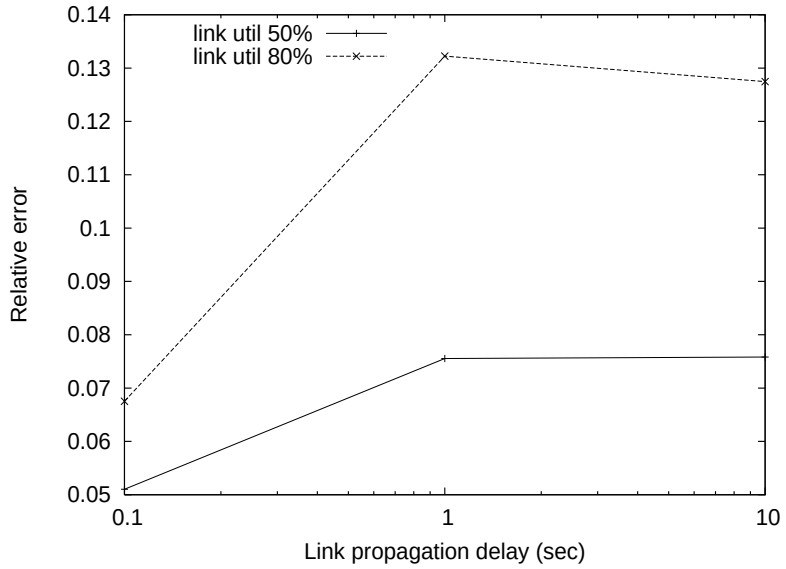


(b) Relative error

Figure 4.8: Results on Unconstrained Rate Smoothing



(a) Ratio of flow rate changes and execution time (simulation with constrained rate smoothing / simulation without rate smoothing)



(b) Relative error

Figure 4.9: Results on Constrained Rate Smoothing

4.3 Integration with Packet-Oriented Flows

In this section, we discuss how to integrate fluid-oriented and packet-oriented flows at the same port. This is illustrated in Figure 4.10. The interaction between fluid-oriented and packet-oriented flows is mutual: when they compete for both bandwidth and storage in the buffer, traffic represented by fluid-oriented models affects traffic represented by packet-oriented models and vice versa. Ideally, the resource allocation at the multiplexing point should be fair; that is to say, the behavior of the traffic should be transparent to the way in which it is represented. In our approach, we model the effect that fluid flows have on packet flows separately from the opposite effect. These models are discussed in the following subsections.

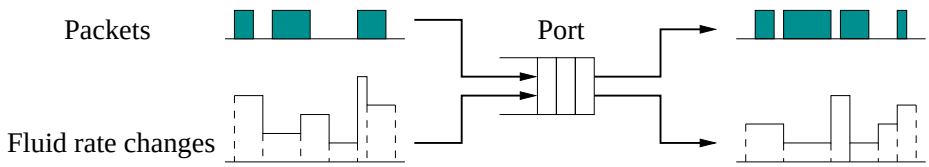


Figure 4.10: Integration of Packet And Fluid Flows

4.3.1 Effect of Packet Flows on Fluid Flows

At a port that handles traffic mixing packet flows and fluid flows, we treat the aggregate packet flow as a special fluid flow: it arrives at the port with an input rate that is equivalent to the aggregate rate of packet flows but immediately disappears into a sink after departing from the port. With addition of this special fluid flow, we are able to apply the event-driven fluid simulation as described in Section 4.2.1 directly except that it is unnecessary to compute departure rates for the special fluid flow. The rate smoothing technique discussed in Section 4.2.3 can still be applied to mitigate the ripple effect that occurs to regular fluid flows.

The remaining question, then, is how to estimate the arrival rate of the aggregate packet flow. An ideal estimation mechanism should satisfy the following principles:

- The estimated rate of the aggregate packet flow should be sufficiently accurate so that the competition for bandwidth and storage in the buffer between the aggregate packet flow and the regular fluid flows is “fair”. Fairness here means that the final service rates and buffer occupancy are relatively insensitive to the way in which the traffic is represented.

- The computation cost on estimating the rate of the aggregate packet flow should be as low as possible. From this perspective, the naive solution that fluidizes each packet arrival with two rate changes, one corresponding to its starting bit and the other to its ending bit, is too computationally costly.

- The estimated rates of the aggregate packet flow should not change too frequently. In a congested port, frequent changes on the estimated rates lead to a lot of computation on deriving the departure rates for regular fluid flows.

In order to estimate the input rate of the aggregate packet flow, we define an adjustable measurement window W. When a packet with b bytes arrives at time t after no packet appears during time interval $[t-W, t)$, we set the instantaneous rate of the aggregate packet flow as b/W, and then schedule a *packet rate estimation timer* that fires at time $t+W$. As simulation time elapses, other packets may arrive before the timer fires. We use B to denote the total number of bytes that arrive within time interval $(t, t+W]$. Then, when the packet rate estimation timer fires at time $t+W$, the instantaneous rate of the aggregate packet flow is updated as B/W, and we decide whether to reschedule the packet rate estimation timer in the following way: if B is not zero, the packet rate estimation timer is rescheduled to fire after another measurement window elapses, or otherwise, the timer is simply canceled. Note that when the rate of the aggregate packet flow changes, there exists a delay before it is detected by the above estimation method. The selection on the measurement window W reflects a tradeoff between accuracy and efficiency. If a larger measurement window is used, the packet rate estimation timer fires relatively less frequently, thus causing less computation cost, but the estimation method is less responsive to the rate changes of the aggregate packet flow. In the implementation, we use an adjustable measurement window. When the packet rate estimation timer fires, the next measurement window spans the time for approximately k packet arrivals, based on the current estimate on the rate of the aggregate packet flow. In addition, in order to avoid a too large or too small measurement window, an upper bound W_{max} and a low bound W_{min} are imposed on W. In brief, the measurement window W is updated as follows:

$$W = max\{min\{kl/\lambda_p, W_{max}\}, W_{min}\}, \tag{4.8}$$

where λ_p is the current estimate on the rate of the aggregate packet flow.

4.3.2 Effect of Fluid Flows on Packet Flows

Recall that in the implementation described in Section 4.2.1, each fluid port has a state variable to keep the current backlog in the buffer. When mixed traffic representations are supported, we fluidize the aggregate packet flow and the backlog is updated as if the aggregate packet flow is a regular fluid flow. Upon a packet arrival, the backlog as calculated can be used to determine whether the packet should be dropped due to buffer overflow, and if there is enough vacancy in the buffer, how much waiting time this packet has to suffer.

However, as mentioned before, the rate estimation for the aggregate packet flow has some delay in responding to its rate changes. This may lead to inaccuracy in the backlog computation if the delay is very large. Since the backlog calculated in this way serves the purpose of making decisions on packet dropping and computing queueing delays, the inaccuracy may be amplified. In our approach, therefore, a separate state variable is maintained to keep the current backlog from the perspective of packet flows. Let Ψ_q be this state variable at port q. We use another state variable, t'_l, to store the last time when this backlog is updated.

Another observation is that while a packet is being served, traffic that has arrived during this period time is accumulated in the buffer until the packet has fully departed from the port. Hence, we introduce a state variable t_{res} to denote the residual time the packet being served requires to get fully transmitted. We initialize t_{res} to be 0 in the simulation setup.

Backlog Ψ_q is updated whenever a rate change event fires. Suppose rate change event e fires at port q at simulation time t. We update backlog Ψ_q as in the following two cases:

- **Case** A_1**:** $t - t'_l \geq t_{res}$. This means that at simulation time t, the last packet must have fully departed from port q. Then, the backlog changes differently between time intervals [t'_l, $t'_l + t_{res}$) and [$t'_l + t_{res}$, t). The traffic that arrives in the first one has to wait in the buffer, causing the backlog to increase, but in the second interval, the backlog may drain, accumulate, or remain unaltered, depending on the relationship between the aggregate arrival rate into the port and the link bandwidth associated with the port. Hence, we update the backlog at time t as follows:

  $$\Psi_q(t) = min(\Phi_q, max(0, \Psi_q(t'_l) + \Lambda_q^{(in)}((t'_l)^+) \times t_{res} + (\Lambda_q^{(in)}((t'_l)^+) - \mu_q) \times (t - t'_l - t_{res})). \tag{4.9}$$

  Note that the backlog in the buffer should not be negative or exceed the buffer size. After computing the new backlog, we update t_{res} to be 0 and t'_l to be t.

- **Case** A_2**:** $t - t'_l < t_{res}$. This means that at simulation time t, the last packet has not fully left port q yet. Traffic that arrives after t'_l has to wait in the buffer. Hence, we update the backlog as follows:

  $$\Psi_q(t) = min(\Phi_q, \Psi_q(t'_l) + \Lambda_q^{(in)}((t'_l)^+) \times (t - t'_l). \tag{4.10}$$

  After computing the new backlog, we decrease t_{res} by $t - t'_l$ and update t'_l to be t.

Now we discuss how to handle a packet arrival into a port. Consider a packet with length l_p arrives at port q at simulation time t. First, we update the backlog in the buffer in the same way as processing a rate change event and let t_{res} decrease by $t - t'_l$. Based on the current backlog, we then process the new packet as follows:

- **Case** B_1**:** $\Psi_q(t) = 0$ and $t_{res} = 0$. In this case, there is no backlog in the buffer and no packet is being served. Hence, the packet is served at the instant time when it arrives. We set t_{res} to be l_p/m_q, which is the transmission time of the new packet. The new packet does not suffer any delay, and it departs from the port entirely at time $t + l_p/m_q$.

- **Case** B_2**:** $\Psi_q(t) = 0$ and $t_{res} > 0$. In this case, there is no backlog in the buffer but a packet is being served by the port. Hence, the new packet has to wait until the previous one is fully served. We then put the new packet into the buffer by resetting $\Psi_q(t)$ to be l_p. The exact time when the new packet fully departs from the port is $t + t_{res} + l_p/m_q$. That is to say, the new packet leaves the port after the time for the packet currently being served to finish its transmission plus its own transmission time.

- **Case B_3:** $\Psi_q(t) > 0$ and $\Psi_q(t) + l_p \leq \Phi_q$. In this case, there is backlog in the buffer, and the new packet has to wait for the transmission of itself until this backlog drains completely. If there is a packet being served, it also has to wait for its residual service time. Moreover, there is enough vacancy in the buffer to accommodate the new packet. Then, we add l_p to the current backlog $\Psi_q(t)$. Similarly, the exact time when the new packet fully departs from the port is $t + t_{res} + \Psi_q(t)/\mu_q$ (note that the transmission time of the new packet has already been counted because $\Psi_q(t)$ includes l_p).

- **Case B_4:** $\Psi_q(t) > 0$ and $\Psi_q(t) + l_p > \Phi_q$. In this case, the new packet cannot be served at the instant time when it arrives at the port because there is backlog in the buffer. Furthermore, the calculated backlog does not leave enough storage space to hold the new packet. A naive solution is to simply drop the packet. This, however, does not conform to the reality because even at an overloaded port, there is some probability for a fraction of traffic to get into the buffer. Let $\lambda_p(t)$ and $\lambda_f(t)$ denote the aggregate packet arrival rate and the instant fluid arrival rate at simulation time t respectively. When the newly arrived packet cannot be accommodated in the buffer, we drop the packet with probability

  $$P_{loss} = 1 - min(1, \frac{\mu_q}{\lambda_p(t) + \lambda_f(t)}). \tag{4.11}$$

  The packet loss probability formed as above can be explained as follows. If the port is overloaded when the packet arrives (i.e., $\lambda_p(t) + \lambda_f(t) > \mu_q$), the overloaded portion is dropped; otherwise, the packet is put into the buffer and the backlog is set equal to the buffer size. In addition, we ensure that the packet dropping probability is a nonnegative. From a lot of experimentation, though, we find that the packet dropping probability presented in Formula (4.11) is slightly lower than that in the equivalent packet-oriented simulation. Therefore, we add a constant scaling factor to compensate for this difference. The packet dropping probability in our implementation is

  $$P_{loss} = 1 - \beta \times min(1, \frac{\mu_q}{\lambda_p(t) + \lambda_f(t)}), \tag{4.12}$$

  where β is 0.9 in all the experiments discussed later.

An important problem with the packet dropping probability in Formula (4.12) is how to determine the current aggregate packet rate $\lambda_p(t)$. In section 4.3.1, we have given an approach to estimating the aggregate packet arrival rate for bandwidth allocation among packet flows and regular fluid flows. In order to balance the computation cost and responsiveness to the rate change of real packet flows, in that method we have adopted an adaptive measurement window that spans the time for a constant number of packet arrivals. If the packet rate measured in that method is applied in Formula (4.12) for estimating packet loss probability, the packet loss probability behaves very sensitively with respect to the measurement window. If the window is relatively large, the real aggregate packet arrival rate may differ significantly from the current estimate and the derived packet dropping probability is imprecise. If an overly small measurement window is applied, not only high computation cost is required to measure the packet rate, but also the estimated packet rate may be unreasonably high, thus causing an overestimation of the packet loss rate.

In order to achieve a robust and accurate estimation on the current packet arrival rate, we apply the approach used in [135] to measuring the arrival rate of the aggregate packet flow in Formula (4.12). Suppose a packet of length l arrives at time t. Let t_{last} be the time when the previous packet arrives. We use λ_p^{last} be the packet flow arrival rate estimated when the last packet arrived. The new packet flow arrival rate is then estimated by

$$\lambda_p^{new} = (1 - e^{-\tau/K})\frac{l_p}{\tau} + e^{-\tau/K}\lambda_p^{last}, \tag{4.13}$$

where τ is the last packet interarrival time (i.e., $t - t_{last}$) and K is a constant. Compared with the aforementioned window-based rate estimation method, this one is able to capture the most recent packet arrival rate intantaneously (the first item on the right side of Equation (4.13)). On the other hand, this approach also distinguishes itself from the normal exponential averaging method in which a constant exponential weight is used. In [135], it has been observed that the normal approach behaves relatively sensitive to the packet length distribution and the derived packet rate estimate differs from the real rate by a factor. Instead, the approach with weight $e^{-\tau/K}$ avoids this problem. Some guidelines on selecting K have been given in the same paper. For example, K should be large enough to dampen the effect that packet delay-jitters have on the estimation process; on the other hand, a smaller K responds better to the rapid packet rate fluctuations. In the experiments discussed later, we let K be 100 ms.

Note that the original approach is used to estimate the arrival rate of an individual flow. In such a context, it is ensured that the packet interarrival time is at least no less than the packet transmission time. Therefore, the estimated rate of a flow never exceeds the bandwidth of the link associated with the input port where the packet flow arrives. However, our problem requires estimating the aggregate packet flow rate at an output port. It is thus possible that multiple packets arrive at an output port at the same time or within a very small time frame in the simulation, and the derived estimate on the aggregate packet flow arrival rate is too high. In order to solve this problem, we estimate the aggregate packet flow arrival rate from each input port at every output port. It is thus guaranteed that the estimated rate of the aggregate packet flow from a particular input port does not exceed the bandwidth of the link associated with that port. In addition, packets may also be originated from the local machine. Hence, we also estimate the arrival rate of the packet traffic sent by the local machine. The aggregate packet flow rate estimate used in Equation (4.12) is then obtained by summing the arrival rate estimates of packet-oriented traffic from the local machine and all the input ports.

4.3.3 Simulation Results

In this section we evaluate our model with simulation experiments. The topologies used in these experiments involve the classical "dumbbell" topology shown in Figure 4.11 and the more complex ATT backbone network shown in Figure 4.7. Given a particular topology, we change some of its configuration parameters or rates at which flows inject traffic into the network. The accuracy and performance of our approach are studied under varied network configurations.

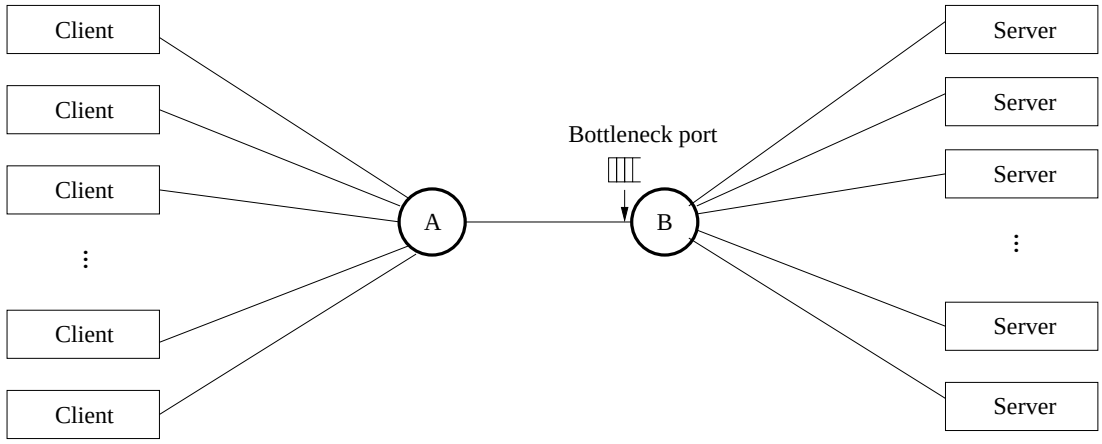


Figure 4.11: Dumbbell Topology

4.3.3.1 Experiment 1: Dumbbell Topology with MMFM Traffic

In this part, we present the simulation results with the dumbbell topology depicted in Figure 4.11. In this topology, a set of clients are connected to router A and a set of servers are connected to router B; router A and router B are directly connected by a bottleneck link. The links in the topology are homogeneous – they have the same bandwidth and propagation delay. The buffer size at the bottleneck port is set to be the bandwidth-delay product of the bottleneck link.

We vary the number of client-server pairs in the topology. One set of experiments has 1 TCP stream, 1 UDP stream, and 10 background flows. In the TCP stream, the TCP client requests a file of 5M bytes from the corresponding TCP server. After the TCP server has successfully delivered the requested number of bytes, the TCP client "sleeps" for an exponentially distributed period with mean of 5 seconds. After the off period, the TCP client initiates another transfer of the same file size. The process repeats itself until the simulation is over. In the UDP stream, the UDP client requests a file of 5M bytes from its peer server. When the UDP server receives the request, it transfers the file to the client at a constant rate. In the experiments, we let the rate be equal to the 10% of the bottleneck link bandwidth. In addition, when the request is initiated, the UDP client schedules a user timer, whose fire time is set slightly later than the time for the server to finish the data transfer. When the user timer fires, the UDP client remains "silent" for an exponentially distributed period with mean of 5 seconds. After the off period, the UDP client sends another file transfer request to its peer server. This process repeats itself until the simulation terminates. The background flows are unidirectional from the servers to their peer clients. In the hybrid simulation, we use the MMFM traffic model to generate fluid rates for each background flow. Each MMFM traffic source is an *on/off* process: in the "on" period the source sends traffic at a constant rate, and in the "off" period it does not send traffic. We fix each MMFM traffic source's emission rate in the *on* period to be 2Mbps; the mean times of both the "on" state and the "off" state are 1 second. In the pure packet-level simulation, the

MMFM traffic model is replaced with an equivalent packet-oriented model which generates packets at the same rates in its fluid-oriented counterpart. In the experiments, we vary the link bandwidth between 12.5Mbps, 20Mbps, and 50Mbps; correspondingly, the traffic loads contributed by the background traffic flow are 20%, 50%, and 80% of the bottleneck link bandwidth. We also vary the link propagation latency between 5 ms, 50 ms, and 500 ms.

We measure the packet loss probability at the bottleneck port. Figure 4.12 describes the results with 95% confidence interval. The ranges are very small and we thus hardly see them in the figure. In the pure packet-level simulation, we collect the packet loss probabilities of foreground traffic flows (i.e., the TCP stream and the UDP stream). Hence, packet loss probabilities of the same portion of traffic are compared between the pure packet-level simulation and the hybrid simulation. From the graph, we find that the packet loss probabilities from two simulation approaches match closely. When the bottleneck bandwidth is 12.5Mbps, the average relative error on the packet loss probabilities is 4.7% with standard deviation 2.6%; when the bottleneck bandwidth is 20Mbps or 50Mbps, the packet loss probabilities are close to 0, which makes the derived relative errors hardly meaningful.

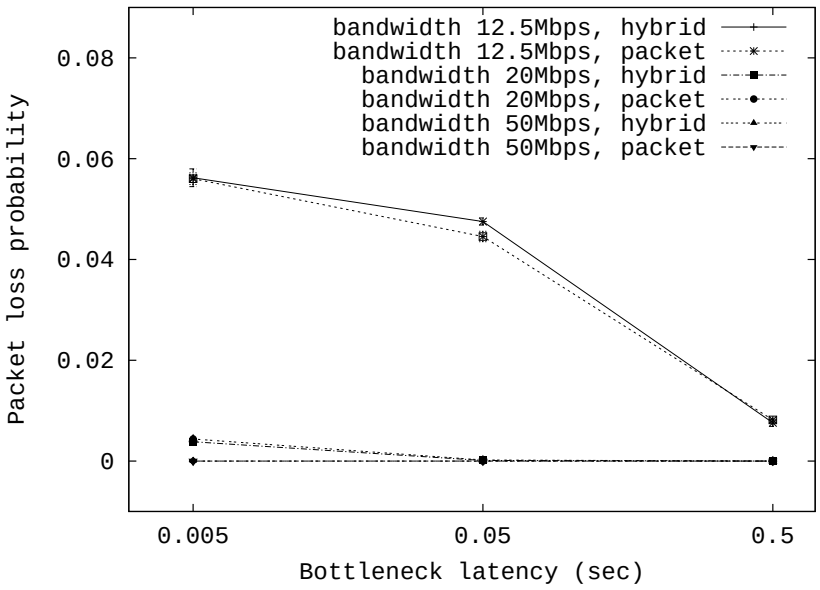


Figure 4.12: Packet Loss Probability under 10 Background Flows

Figure 4.13 gives the average *goodput* of TCP streams. The goodput of a TCP stream is defined to be the average number of bytes successfully delivered per simulation time unit using TCP protocol. The average relative errors on the TCP goodputs corresponding to the bottleneck bandwidth are 20.9% with standard deviation 9.3% at 12.5Mbps, 6.6% with standard deviation 6.4% at 20Mbps, and 0.14% with standard deviation 0.22% at 50Mbps. It is clear that a higher bottleneck

link bandwidth causes less errors. From Figure 4.12, we know that the packet loss probability on the bottleneck link is very small when its bandwidth is high, and therefore, the complicated congestion control mechanism in TCP is seldom triggered. This explains the decreasing relative errors when bottleneck bandwidth increases. In Figure 4.14, we present the results on the average round trip time for the TCP stream. An excellent agreement is achieved because the relative error on every simulation configuration is below 1%.

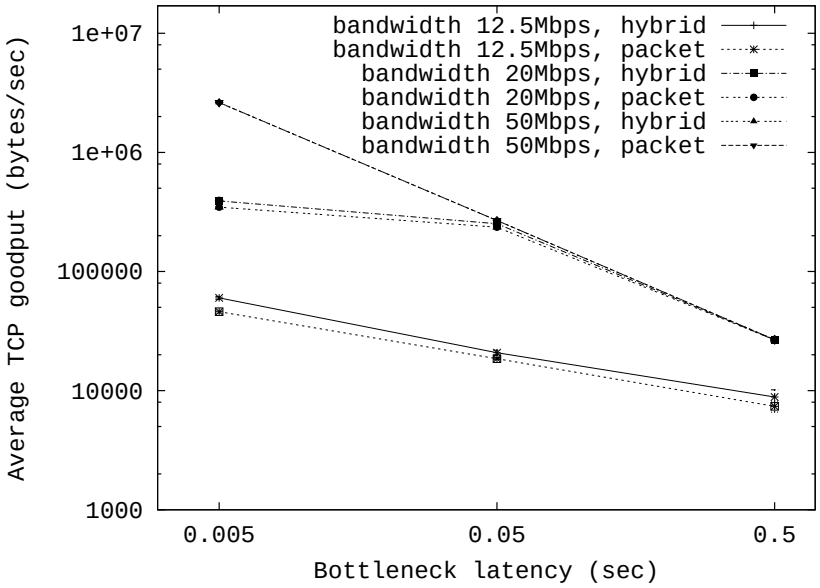


Figure 4.13: TCP Goodput under 10 Background Flows

Figure 4.15 describes the simulation results on how much traffic has been successfully delivered by the UDP protocol. It is clear that the pure packet-level simulation and the hybrid simulation produce very close results. The relative error on every simulation configuration is less than 1%.

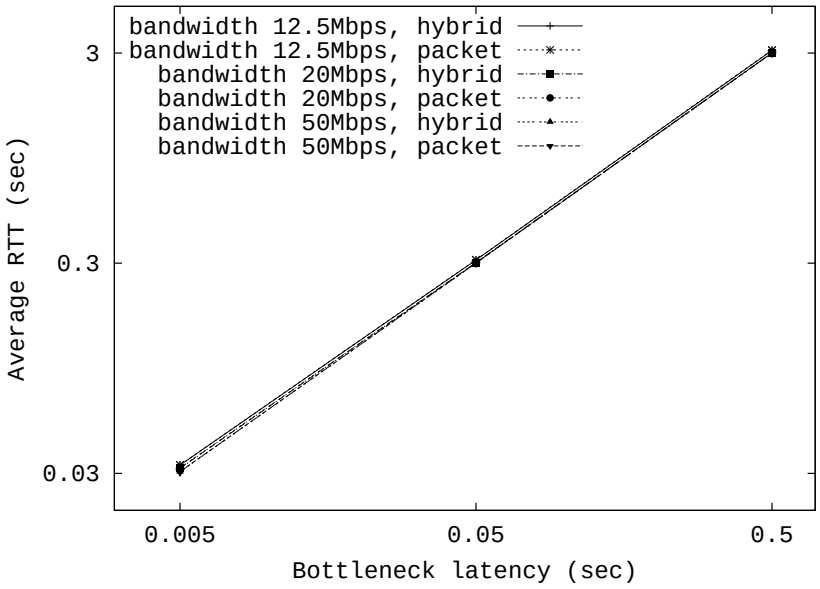


Figure 4.14: TCP Round Trip Time under 10 Background Flows

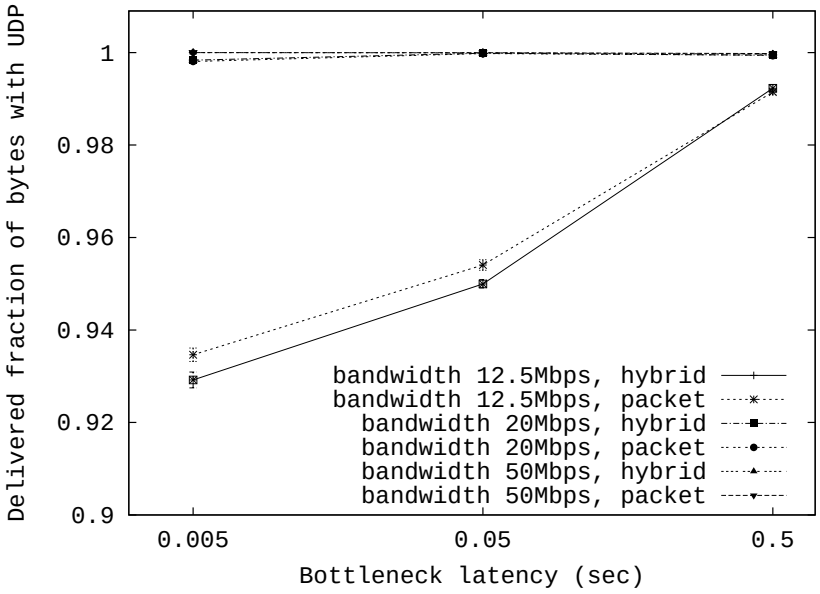


Figure 4.15: Delivered Fraction of UDP Traffic under 10 Background Flows

In the second set of experiments, we increase the number of both foreground and background flows, but the ratio of the former to the latter remains the same as in the first set. In these experiments, there are 10 TCP streams, 10 UDP streams, and 100 background flows. The traffic pattern of each TCP stream is unaltered. The traffic pattern of a UDP stream is also the same, except that the sending rate of each UDP server is only one tenth of the rate in the first set. Hence, the overall peak traffic load on the bottleneck link contributed by UDP traffic is still 10% of the bottleneck link bandwidth. Similarly, a background flow from a server to its peer client is also modeled by a MMFM model in the hybrid simulation, and a packetized MMFM model in the pure packet-level simulation; when an MMFM source is in the "on" state, its sending rate is 0.2Mbps. Therefore, the average traffic load of the background traffic on the bottleneck link is 20% at 12.Mbps, 50% at 20Mbps, and 80% at 50Mbps.

As before, we measure the packet loss probability at the bottleneck port. The simulation results are depicted in Figure 4.16. The average relative errors corresponding to the bottleneck link bandwidth are 4.6% with standard deviation 0.3% at 12.5Mbps, 16.4% with standard deviation 3.0% at 20Mbps, and 33.7% with standard deviation 8.5% at 50Mbps. We have noticed that the high relative error under high bottleneck link bandwidth results from the relatively small packet loss probability in this range.

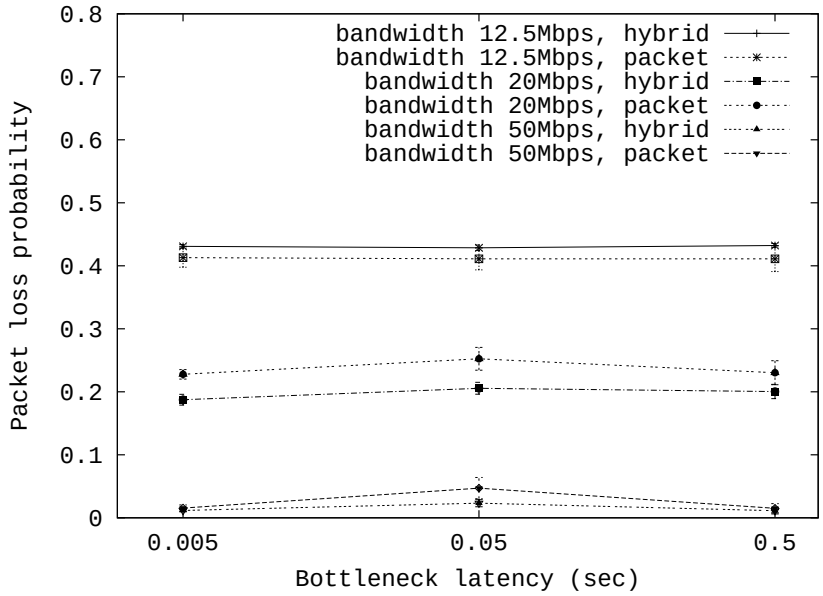


Figure 4.16: Packet Loss Probability under 100 Background Flows

The average TCP goodputs under varied network configurations are described in Figure 4.17. The average relative errors corresponding to bottleneck link bandwidth are 16.45% with standard deviation 6.8% at 12.5Mbps, 32.5% with standard deviation 16.1% at 20Mbps, and 16.5% with

standard deviation 3.3% at 50Mbps. Compared with the results under 10 background flows, the relative errors under 100 background flows are relatively higher. Note that although we scale the sending rate of the UDP servers and the background traffic flows so that they offer the same loads on the bottleneck link, there is much heavier TCP traffic load on that link under 100 background flows. Because of the increased overall traffic load on the bottleneck link, TCP behaves more dynamically when responding to the relatively higher congestion in the network. This explains the increased relative errors on TCP goodputs under 100 background flows. In addition, we also measure the round trip times of the TCP packets. The results are shown in Figure 4.19. The relative errors corresponding to the bottleneck link bandwidth are 1% with standard deviation 0.8% at 12.5Mbps, 0.4% with standard deviation 0.4% at 20Mbps, and 0.8% with standard deviation 0.9% at 50Mbps. Obviously, the two simulation approaches have achieved excellent agreements on the round trip times.

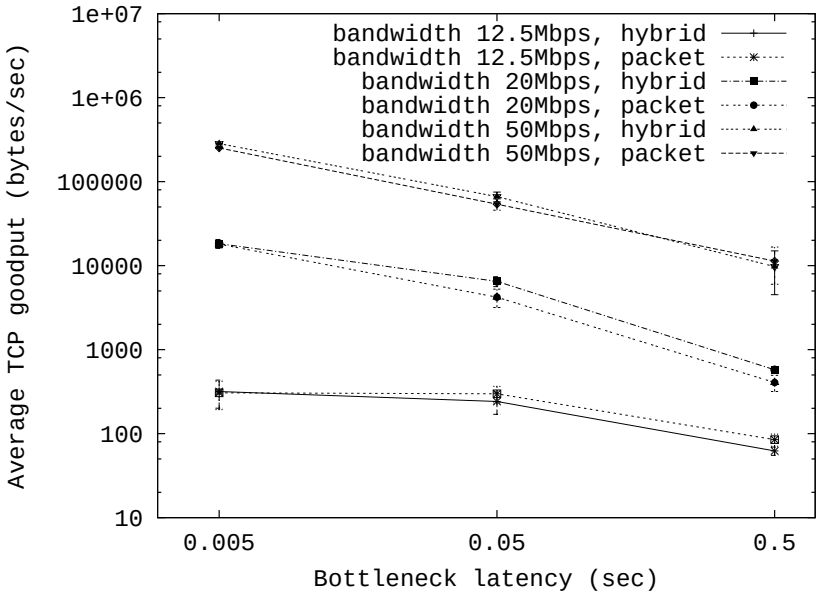


Figure 4.17: TCP Goodput under 100 Background Flows

The accuracy results on UDP traffic under 100 background flows are presented in Figure 4.19. It depicts the successfully delivered UDP traffic in the simulation. The average relative errors corresponding to the bottleneck link bandwidth are 3.2% with standard deviation 0.4% at 12.5Mbps, 5.3% with standard deviation 1.2% at 20Mbps, and 1.3% with standard deviation 1.4% at 50Mbps. These results suggest a close match on this UDP flow metric between the hybrid simulation and the pure packet-oriented simulation.

The execution speedups of hybrid simulation over pure packet-level simulation under varied network configurations are depicted in Figure 4.20. From the results, we see that hybrid simulation

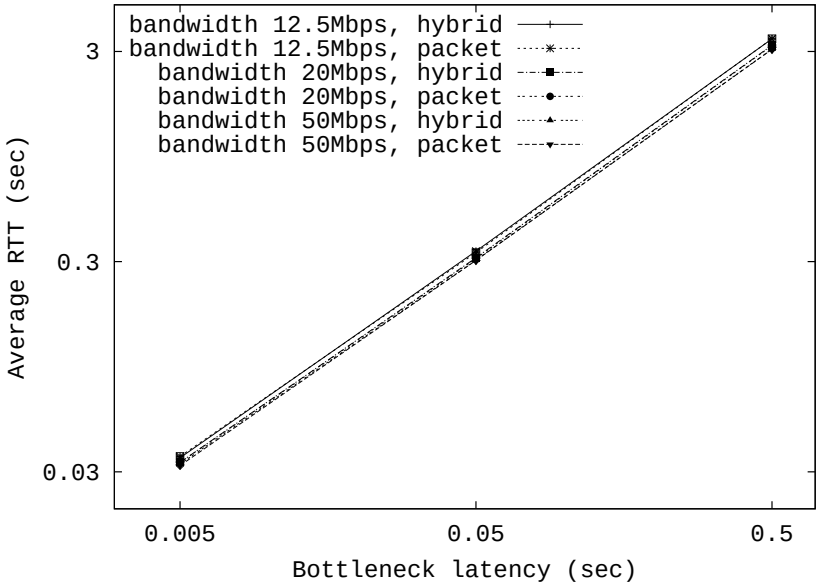


Figure 4.18: TCP Round Trip Time under 100 Background Flows

outperforms pure packet-level simulation under all configurations. The lowest execution speedup is 1.4, when there are 100 background flows and the bottleneck link has bandwidth of 12.5Mbps and propagation latency of 5 milliseconds; the highest execution speedup, which is 90, occurs when there are 10 background flows and the bottleneck link has bandwidth of 50Mbps and propagation latency of 500 milliseconds.

Now we discuss the impact that the network configurations have on the execution speedup of hybrid simulation over pure packet-level simulation. When the bottleneck latency increases and other conditions are the same, the execution speedup gained from hybrid simulation also increases. Since the UDP traffic and the background flows are non-responsive, that is, they do not adapt to the network condition dynamically, changes on bottleneck latency hardly affects their traffic. Rather, TCP traffic is sensitive to the round trip times of packets – its throughput is roughly inversely proportional to the average round trip time [92]. Therefore, increasing the bottleneck latency reduces the fraction of packet-oriented TCP traffic in hybrid simulation; hence, it is able to achieve a higher execution speedup against the pure packet-level simulation.

The effects of the varied bottleneck bandwidth on the performance gain from the hybrid simulation are mixed when other conditions are the same. First, as the bottleneck bandwidth increases, the packet loss probability at the bottleneck port decreases and thus more packet-oriented traffic is able to get through the port. From this perspective, the pure packet-level simulation should be outperformed by the counterpart hybrid simulation. Second, increasing the bottleneck bandwidth also increases the fraction of packet-oriented foreground traffic. Recall that the offered load of UDP traf-

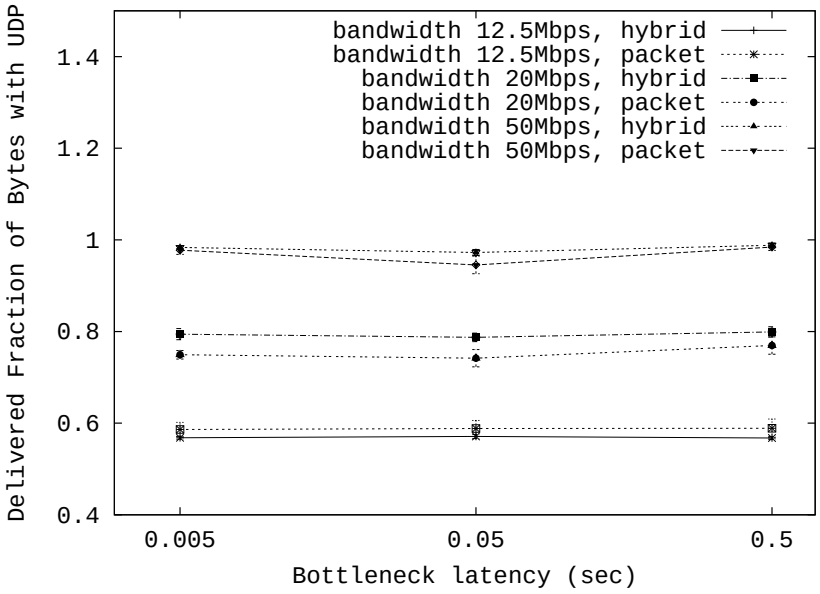


Figure 4.19: Delivered Fraction of UDP Traffic under 100 Background Flows

fic on the bottleneck link is 10 percent of its bandwidth. Furthermore, a relatively large bottleneck bandwidth reduces the packet loss probability at the bottleneck port; an increase on the throughput of TCP stream results because it is roughly inversely proportional to the square root of the packet loss rate [92]. Third, increasing the bottleneck bandwidth also affects the interaction between fluid flows and packet flows in the hybrid simulation. A low bottleneck bandwidth causes relatively more congestion. Under congestion, changes on the aggregate packet flow rate trigger computation of departure rates of fluid flows. On the other hand, heavy congestion also causes more packet losses, which demands more time on computing the packet loss probability as described in Equation (4.13) in hybrid simulation. These reasons result in mixed effects that varying bottleneck bandwidth imposes on the performance gain from the hybrid simulation. In most cases, the first factor dominates over the other two because the background flows contribute to the most significant fraction of the whole traffic. However, some exceptions occur when the bottleneck latency is small. In this range, the last two factors play a more important role than the first one, and the effect of increasing the bottleneck bandwidth on the performance gain of hybrid simulation is therefore slightly negative.

When we increase the number of flows (both foreground and background flows) in the simulation, it is clear that the relative execution speedup achieved by hybrid simulation over pure packet-level simulation drops. Although we configure the network in a way so that the offered load on the bottleneck link from background flows and packet UDP flows is maintained at the same level, the TCP traffic load under 100 background flows is heavier than that under 10 background flows. Since the fraction of packet-oriented traffic in hybrid simulation is higher in the former case, lower relative

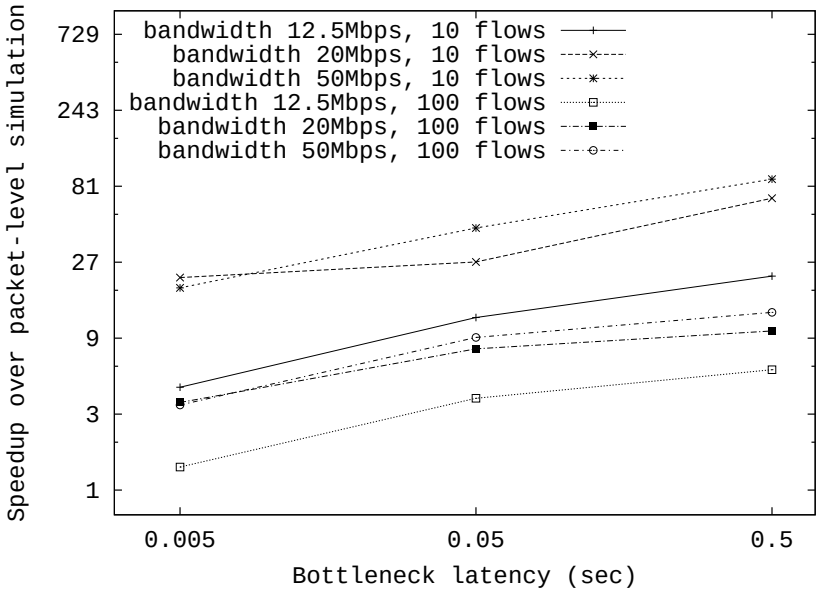


Figure 4.20: Speedup of Hybrid Simulation over Pure Packet-Level Simulation under Dumbbell Topology

execution speedup can be achieved from hybrid simulation. On the other hand, the performance of fluid-oriented traffic simulation depends on the number of flows. When there are 100 fluid-oriented background flows in the hybrid simulation, more computation workload is inevitably needed to propagate their flow rate changes than that when there are only 10 fluid-oriented background flows. All these explain the observation that the relative execution speedup of hybrid simulation over pure packet-level simulation diminishes as the number of flows in the network increases.

4.3.3.2 Experiment 2: ATT Backbone with MMFM Traffic

In this section, the realistic ATT backbone network depicted in Figure 4.7 is used to study the accuracy and performance of our approach to handling mixed traffic representations. The background flows in the topology are "*all-to-all*": for every pair of routers in the topology, there exists a background flow whose ingress traffic is modeled by the MMFM source model. As in previous experiments, the model is actually an *on/off* process; both the "on" state and the "off" state have mean duration of 1 second. The emission rate in the "on" state is the same for all the background traffic sources. We adjust this rate to achieve two average background traffic loads on each link, 50% utilization and 80% utilization.

In the original topology, we attach an end host to every router. Every end host has installed a TCP client, a UDP client, a TCP server, and a UDP server on it. In each experiment setting,

every TCP client randomly chooses a TCP server from all the end hosts but itself, and requests a file transfer of size 5M bytes from the selected server. When the client receives the requested file, it remains "silent" for an exponentially distributed duration with mean of 5 seconds; after waking from the "silent" period, it initiates another request to the same server it has chosen. The above process repeats until the simulation finishes. Similarly, in each experiment setting, every UDP client randomly chooses a UDP server from all other end hosts and requests a file transfer of 5M bytes from there. After receiving a request, a UDP server sends back the requested bytes at the rate of 5Mbps. When a UDP client sends out its request, it schedules a user timer which fires slightly later than the time when the corresponding server finishes its transfer. When the user timer fires, the UDP client goes to the "silent" period, which also conforms to an exponential distribution with mean of 5 seconds.

We randomly generate 10 experiment settings, in which each TCP client randomly selects a TCP server and a UDP client randomly selects a UDP server. In every experiment, there are 27 TCP streams and the same number of UDP streams, all represented with packet-oriented models. We measure the average goodput of the 27 TCP streams and the average delivered fraction of UDP traffic throughout the simulation. The results with 95% confidence interval on the average TCP goodputs are given in Figure 4.21. The x-axis represents the 10 experiment settings. The average relative errors corresponding to the average background traffic loads are 6.3% with standard deviation 2.0% at 50% and 3.4% with standard deviation 1.7% at 80%. The results on the average delivered fraction of UDP traffic under the 10 experiment settings are given in Figure 4.22. The relative errors corresponding to the average background traffic load are 0.6% with standard deviation 0.3% at 50% and 0.9% with standard deviation 0.8% at 80%. All these results suggest that pure packet-level simulation and hybrid simulation produce well-matched results on the flow metrics discussed.

We present the relative speedup of hybrid simulation over pure packet-level simulation in Figure 4.23. We run each experiment setting for 3600 seconds, and each simulation run is replicated for 10 times. From the graph, we have observed that when the average background traffic load is 50% of the link bandwidth, the relative execution speedup is approximately 4; when the average background traffic load is 80% of the link bandwidth, the relative execution speedup is approximately 8. The difference between these two numbers can simply be explained as follows. If background traffic load in the network is heavier, then a relatively larger portion of the whole traffic is as fluid flows, which thus brings better relative speedup from the hybrid simulation.

The average execution times under two background traffic loads are given in Table 4.4. It is clear that the pure packet-level simulation cannot be accomplished in real time, but the hybrid simulation is finished within less than half the real time. This suggests that pure packet-oriented traffic simulation is inappropriate in cases where meeting real time constraint is of crucial significance. In addition, with increasing background traffic load, the execution time by hybrid simulation decreases. This may be counter-intuitive because a higher fluid traffic load in the network usually leads to heavier congestion and thus more computation on fluid rate propagation. In the hybrid simulation, however, a higher background traffic load may cause more packet losses because of congestion, thus reducing the overall computation load from this perspective. The experimental results tell us that the simulation performance is affected by the latter factor to a heavier degree.

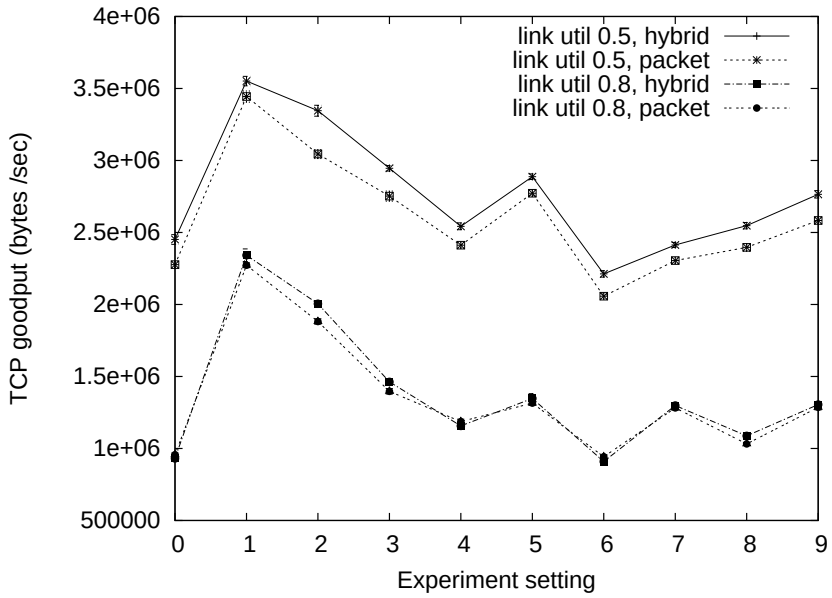


Figure 4.21: TCP Goodput under ATT Topology

Background Traffic Load	Hybrid Simulation	Pure Packet-Level Simulation
0.5	984.85 (sec)	4013.08 (sec)
0.8	639.92 (sec)	5193.92 (sec)

Table 4.4: Average Execution Time for Simulating ATT Topology

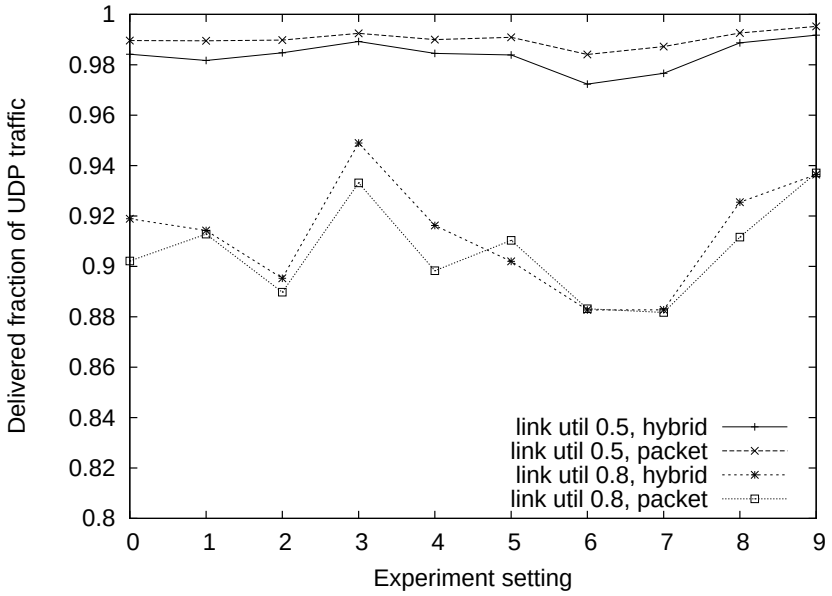


Figure 4.22: Delivered Fraction of UDP Traffic under ATT Topology

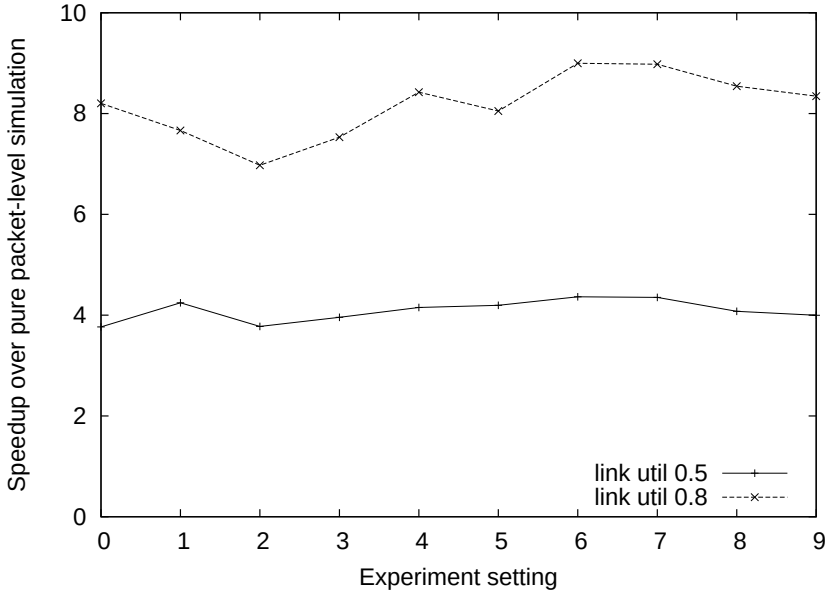


Figure 4.23: Speedup of Hybrid Simulation over Pure Packet-Level Simulation under ATT Topology

4.4 Hybrid Simulation of TCP Traffic

In Section 4.3, we have discussed how to integrate event-driven fluid-oriented traffic simulation and packet-oriented traffic simulation, and in the experiments, the non-responsive MMFM model is used to generate traffic represented as fluid rate changes. In this section, we continue the discussion in the context of TCP traffic simulation. As a large portion of the current Internet traffic uses TCP, it is thus important to understand both how TCP responds to the dynamic network conditions and how its traffic affects network dynamics. Simulation is a tool that is widely applied to achieve these objectives. It is, therefore, a meaningful undertaking to investigate how to accelerate simulation of TCP traffic. In this section, we discuss the possibility of using hybrid simulation, which integrates fluid-oriented and packet-oriented TCP models, to achieve performance improvement. In the hybrid simulation, we have implemented Nicol's fluid-based TCP model [104][108], with some simplifications in order to improve simulation efficiency.

4.4.1 Nicol's Fluid Modeling of TCP

Prior to the description of Nicol's Fluid-Based TCP model, a brief introduction to TCP's flow control and loss recovery mechanism provides necessary background knowledge. TCP's flow control mechanism is centered on a sliding congestion window, whose size is adapted to reflect the current network condition. A TCP agent works in one of two modes: *slow start* and *congestion avoidance* [54]. In the *slow start* mode, TCP sender increases its congestion window size by the size of that data segment after receiving every data segment acknowledged by the receiver. In a topology where the round trip time is large enough to accommodate the transmission of all data segments within a congestion window, the congestion window size doubles every round trip time. The continuing growth of congestion window size at an exponential rate may cause congestion in the network. If a packet loss is detected, or the congestion window size exceeds a threshold called *mode transition threshold*, TCP enters the *congestion avoidance* mode. In this mode, TCP increases its congestion window size much more slowly compared with how it does in the *slow start* mode: the congestion window size is expanded by only one data segment after the volume of data segments equal to the current congestion window size has been acknowledged by the receiver. Hence, in a topology where the round trip time is large enough to accommodate the transmission of all data segments in a congestion window, the congestion window size increases by a single data segment every round trip time.

TCP's response to congestion in the network is manifold. The simplest approach existing in all TCP variants is to use retransmission timers: if a packet sent out has not been acknowledged when a retransmission timer scheduled for it fires, the packet is assumed to be lost in the network and TCP reacts as follows: it sets its mode transition threshold as half of the current congestion window size and its congestion window size as a single data segment, then re-enters the slow start mode. Relying on only retransmission timeouts for loss recovery, sometimes, severely limits the throughput of the TCP traffic, especially in a network that is only slightly congested. Hence, a fast retransmission mechanism [56] is adopted in many TCP variants. In this mechanism, when more than three duplicate acknowledgement packets are received in a row, a packet is assumed to

be dropped by the network. However, TCP variants differ in how to recover from packet losses detected from a fast retransmission. TCP Tahoe reacts to such a packet loss in the same way as that after a retransmission timeout. In TCP Reno, a fast recovery mechanism [55] is introduced: after a fast retransmit, TCP reduces its current congestion window size by half and uses new incoming duplicate ACKs to expand its congestion window size; after half a window of duplicate ACK packets are received, it increases the congestion window size by one MSS (Maximum Segment Size) and transmits a data segment after each additional ACK packet is received. Besides TCP Tahoe and TCP Reno, other variants like TCP New-Reno and SACK further aim to improve the performance of TCP protocol under varied circumstances. Among all these variants, TCP Reno is the most widely deployed because it has been implemented in the popular BSD 4.3 operating system. All the experiments in this thesis regarding packet-oriented TCP protocol use the Reno variant.

In the following paragraphs, we introduce Nicol's fluid modeling of TCP from three aspects: how to update the emission rate from the TCP sender, how to schedule timers, and how to respond to packet losses in the network.

4.4.1.1 Sending Rates

In Nicol's fluid-based TCP model, a data transfer governed by TCP protocol is abstracted into a fluid stream of bytes, indexed from 0 in byte sequence order. At the TCP sender side, two state variables regarding the current transfer status are maintained: $LBS(t)$ and $LBA(t)$ keep the number of bytes that have been sent and the number of bytes that have been received respectively up to simulation time t. It is obvious that at any simulation time $LBS(t)$ should be no less than $LBA(t)$. We have mentioned that TCP's flow control mechanism is centered on its sliding congestion window and the mode transition threshold. Their values at simulation time t are denoted by $cwnd(t)$ and $ssthresh(t)$ respectively.

Similar to the MMFM traffic model, Nicol's fluid-based TCP model also injects discrete fluid rate changes into the network. Let $\lambda_{send}(t)$ be the data rate sent from the TCP sender at simulation time t, and $\lambda_{ack}(t)$ be the acknowledged byte rate received from the network at simulation time t. Both $\lambda_{send}(t)$ and $\lambda_{ack}(t)$ are piece-wise constant rate functions of simulation time t. One important goal of the model is to determine $\lambda_{send}(t)$, that is, the data transmission rate from the TCP sender. First, if the current congestion window is not filled (i.e., $LBS(t) - LBA(t) < cwnd(t)$), the TCP sender can send out data as fast as it can. The maximum emission rate is constrained by both the data rate from the upper application, denoted by $\lambda_{app}(t)$, and the available transmission bandwidth, denoted by $\lambda_{bw}(t)$. Second, when the congestion window size is tight (i.e., $LBS(t) - LBA(t) = cwnd(t)$), the sending rate is further contingent on both $\lambda_{ack}(t)$, and the rate at which the congestion window size increases. The latter is denoted by $\lambda_{cwnd}(t)$. As we have said, the TCP sender increases its congestion window size differently in two modes. In slow start, the congestion window size increases by one MSS after one data segment has been acknowledged. Therefore, the rate at which the congestion window changes in this mode is equal to the rate at which data segments are acknowledged. That is to say, $\lambda_{cwnd}(t)$ is exactly $\lambda_{ack}(t)$. If the TCP sender is in congestion avoidance mode, we simply let $\lambda_{cwnd}(t)$ be 0, because as we will see later, the congestion window size is changed in a discontinuous manner. Third, it is possible that the current congestion window

size is smaller than $LBS(t) - LBA(t)$. This may occur when the TCP sender shrinks its congestion window size after some packet losses have been detected. In this case, the sending rate is constrained to be 0. In summary, the rule on updating $\lambda_{send}(t)$ is described as follows:

$$\lambda_{send}(t) = \begin{cases} min\{\lambda_{bw}(t), \lambda_{app}(t)\} & \text{if } \Delta(t) < cwnd(t) \\ min\{\lambda_{bw}(t), \lambda_{app}(t), \lambda_{ack}(t) + \lambda_{cwnd}(t)\} & \text{if } \Delta(t) = cwnd(t) \\ 0 & \text{if } \Delta(t) > cwnd(t) \end{cases} \tag{4.14}$$

where $\Delta(t)$ is equal to $LBS(t) - LBA(t)$.

4.4.1.2 Timers

A few timers are necessary for updating the internal state variables in the model. The first one is called *Constrained* timer. It is scheduled when the congestion window size is unfilled (i.e., $LBS(t) - LBA(t) < cwnd(t)$). The purpose of this timer is to notify when the congestion window becomes full. From Equation (4.14), we know that the data transmission rate may need to be changed when the entire congestion window is filled. Suppose that the timer is scheduled at time t. Its fire time, t_c, is then

$$t_c = t + \frac{cwnd(t) - LBS(t) + LBA(t)}{\lambda_{send}(t) - \lambda_{ack}(t) - \lambda_{cwnd}(t)}. \tag{4.15}$$

When the *Constrained* timer fires at time t, the sending rate $\lambda_{send}(t)$ is updated according to the second case in Equation (4.14).

Another timer called *ModeTransition* timer is responsible for mode transition from slow start to congestion avoidance. It is scheduled only when the TCP sender is in slow start mode. When scheduled at simulation time t, its fire time is

$$t_m = t + \frac{ssthresh(t) - cwnd(t)}{\lambda_{cwnd}(t)}. \tag{4.16}$$

As mentioned before, the congestion window size is changed discontinuously in congestion avoidance mode. A timer called *IncreaseCWND* timer is used to notify when the congestion window size should be increased. Let $w_{unack}(t)$ be the amount of unacknowledged data in the congestion window since its size changes last time in congestion avoidance mode. Note that the first time the congestion size changes in congestion avoidance mode is when TCP transmits from slow start mode to congestion avoidance mode. This timer, when scheduled at time t, is to fire at time

$$t_i = t + \frac{w_{unack}(t)}{\lambda_{ack}(t)}. \tag{4.17}$$

In order to avoid firing the *IncreaseCWND* frequently, an optimization is employed in the model. The timer is only running when $LBS(t) - LBA(t) \geq cwnd(t)$ because otherwise it does not affect

the data transmission rate $\lambda_{send}(t)$ based on Equation (4.14). But the current congestion window size needs to be known when the *Constrained* timer is scheduled. It can be reconstructed as follows. Let C be the congestion window size when TCP enters the congestion avoidance mode. Suppose that at the point of calculation, Λ bytes have been acknowledged since the mode transition from slow start to congestion avoidance. We then solve the following equation:

$$\Lambda = x \times C + MSS \times x(x-1)/2. \tag{4.18}$$

Let x' be the positive solution to the above equation. The current congestion window size is then reconstructed as

$$cwnd = C + \lfloor x' \rfloor \times MSS, \tag{4.19}$$

where $\lfloor x' \rfloor$ is the largest integer that is no greater than x'.

4.4.1.3 Loss Recovery

TCP has an adaptive mechanism for responding to packet losses in the network. Nicol's fluid modeling of TCP takes it into consideration as well. In order to analyze lost bytes in the fluid stream, fluid rate changes carry extra information, including the delivered fraction and flow position components. The delivered fraction indicates the fraction of the raw fluid traffic that is successfully delivered; hence, the loss rate of the raw fluid flow can be easily inferred. Flow position components provide information on the current byte sequence that is being delivered. From such information fed back to the TCP sender along with the ACK flows, the amount of lost traffic can be inferred. It is then possible to model the fast retransmit logic in TCP. Besides fast retransmit, Nicol's fluid TCP model also considers retransmission timeouts. The TCP agent fires the retransmission timer every 500 milliseconds. It computes the time at which the current acknowledged byte sequence (i.e., LBA) was transmitted. If that is earlier than the last time the retransmission timer was fired, all the bytes in the current congestion window are retransmitted.

We have discussed Nicol's fluid modeling of the TCP sender. The behavior of the counterpart TCP receiver is relatively simple. One of its primary tasks is to transform the received data byte rates into the acknowledgment byte rates. There may be some discrete information that needs to be passed in a flow. For example, when the TCP sender wants to terminate a flow, it needs to notify the corresponding receiver of it. Such information is carried in extra data structures called *corks*. When a TCP receiver receives a cork, it simply pass it back along with the ACK flow.

4.4.2 Modifications

In some cases, we need to simulate many TCP flows but the high-fidelity behavior of only a small number of them are of interest to us. In such circumstance, we can use a hybrid model to simulate TCP traffic. The TCP flows whose packet-level characteristics are being studied can be represented with packet-oriented models. We can represent the remaining TCP flows with Nicol's fluid TCP model to improve the simulation efficiency. As said in Section 4.4.1, Nicol's fluid TCP model captures rich details of real TCP behavior, including both its window-based flow control mechanism

and its retransmission logic. At the same time, it provides some optimizations to improve its efficiency, such as "lazily" updating the congestion window size when the TCP agent is in congestion avoidance mode. However, we need to make some modifications on it in the hybrid simulation. First, its modeling of TCP's response to packet losses, if detected by fast retransmit logic, is complicated. Furthermore, in order to implement the retransmission timeouts, this model requires that every TCP agent be interrupted every 500 milliseconds to check whether it is necessary to retransmit the entire congestion window. Therefore, although Nicol's model is loyal to the behavior of the real TCP protocol, it can still be simplified to achieve better simulation efficiency. Since the behavior of the fluid-oriented TCP traffic is not of our interest in the hybrid simulation, a precise model characterizing very details of TCP protocol is unnecessary. The bottom line is that the fluid-based TCP model is able to generate the background traffic that is accurate enough for packet-oriented TCP traffic.

From a lot of experimentation, however, we know that the fluid TCP model, if simplified in an improper way, results in significant "unfairness" between fluid-oriented and packet-oriented TCP traffic. For example, if we totally ignore retransmission timeouts in the fluid model and any packet loss detected causes TCP to re-enter transmission mode immediately, then after some congestion occurs on a bottleneck link that packet-oriented and fluid-oriented TCP flows share on their paths, fluid TCP traffic exhibits more aggressive behavior and the packet-oriented TCP flows may thus obtain much less throughput than they should have. On the other hand, totally ignoring fast retransmit is also problematic. Suppose that in the fluid-based TCP, any lost byte is retransmitted when the retransmission timeout occurs. Then in a lightly congested network where most of packet losses are recovered by fast retransmit in reality, packet-oriented TCP flows may obtain a much higher throughput in the hybrid simulation than what they do in pure packet-level simulation.

We have also made another observation from the experiments. This occurs when many fluid-oriented TCP flows traverse through the same congested bottleneck port. Given the FIFO policy as described in Section 4.2.1, the lost traffic spans over every fluid-oriented TCP flow that has a positive input rate. When losses are detected by the corresponding fluid TCP senders, they shrink their congestion window sizes in a way that depends on how they respond to the lost traffic. In the pure packet-level simulation, however, some TCP flows may be so "lucky" that their packets are not dropped when the congestion occurs at the port. If the congestion lasts long enough, these flows may still suffer packet losses, but at a time later than when the congestion starts at the port. Therefore, the fluid-oriented TCP agents in hybrid simulation decrease their congestion window sizes earlier than the packet-oriented TCP agents do in pure packet-level simulation. We call this observation *early window reduction syndrome*. Note that this problem is rooted from the continuous feature of the fluid FIFO port, and thus irrelevant of whether a full-fledged fluid TCP model is adopted. This phenomenon may cause unfairness between packet-oriented and fluid-oriented TCP flows in hybrid simulation. The packet-oriented TCP flows may not suffer losses as the congestion starts. Then fluid-oriented TCP flows deflate their congestion window sizes, leaving packet-oriented TCP flows the opportunity of inflating their congestion windows up to a higher size than in the equivalent pure packet-level simulation.

All the above observations lead us to establish an empirical model on how a fluid-based TCP agent recovers from lost bytes. When the buffer in a port overflows, a special signal is sent to

every fluid-based TCP flows that traverses that port. When such a signal arrives at a TCP sender, say at time t, it immediately strangles existing flows. That is to say, the transmission rate $\lambda_{send}(t)$ is changed to 0. Unlike the original model, the new model eliminates the necessity of detecting when loss finishes. Rather, it sends a special cork, called *loss clearance cork*, along with the latest rate change (i.e., rate 0). This cork traverses through the network, reaches the corresponding TCP receiver, and is finally passed back to the TCP sender. Suppose the cork is returned to the TCP sender at time t'. Let δ_{loss} be the difference between $LBS(t)$ and $LBA(t')$:

$$\delta_{loss} = LBS(t) - LBA(t'). \tag{4.20}$$

It is clear that δ_{loss} is the number of lost bytes because no traffic is sent out since time t. A heuristics is applied here to decide whether the lost traffic is detected by retransmission timeout or fast retransmit by a real TCP protocol. If more packets in the same congestion window have been dropped in the network, it is relatively less likely that the packet losses can be recovered from the fast retransmit and fast recovery mechanism in TCP Reno, and thus more possibly, lost packets are retransmitted when the retransmission timer fires. If only slight congestion occurs in the network, it is relatively likely that the packet losses are recovered from the fast retransmit and fast recovery mechanism. Based on such heuristics, we use the following rule to decide the way in which the lost traffic is retransmitted: if δ_{loss} is larger than a MSS, lost bytes are recovered at retransmission timeouts, or otherwise, they are recovered from fast retransmit and fast recovery mechanism. With this heuristics, if the network is only slightly congested and some fluid TCP flows thus mistakenly receive traffic loss signals from the network, they are able to recover their original congestion window sizes after only a few round trip times, instead of suffering the long retransmission timeouts and the slow start phase.

In the modified model, we do not fire the retransmission timer every 500 milliseconds. Rather, a retransmission timer is scheduled by a TCP agent only when the recovery from some lost traffic attributes to retransmission timeouts. If a loss clearance cork arrives at the TCP sender and δ_{loss} is found to be greater than a MSS, a retransmission timer is scheduled to fire after a particular delay $Timeout$. Then, it comes to how to calculate $Timeout$. The TCP protocol derives the retransmission timeout with the following equations [28]:

$$\begin{cases} DIFF = Sample\_RTT - Old\_RTT \\ Smoothed\_RTT = Old\_RTT + \alpha \times DIFF \\ DEV = Old\_DEV + \beta(|DIFF| - Old\_DEV) \\ Timeout = Smoothed\_RTT + \eta \times DEV \end{cases} \tag{4.21}$$

where $Sample\_RTT$ is the sampled RTT, Old_RTT is the old estimate on the mean of RTT, $Smoothed\_RTT$ is the new estimate on the mean of RTT, Old_DEV is the old estimate on the standard deviation of RTT, DEV is the new estimate on the standard deviation of RTT, and $Timeout$ is the retransmission timeout. In 4.4 BSD Unix, α, β, and η are 1/8, 1/4, and 4 respectively. In order

to obtain an approximate estimate on $Timeout$ in the fluid-based TCP model, the TCP sender needs to sample the current RTT. This is done by sending *RTT sampling corks* along with rate changes. When such a cork is sent, the TCP sender notes down the simulation time at that moment. There is at most one RTT sampling cork on the fly. That is to say, before a RTT sampling cork already sent out is returned to the TCP sender, no other RTT sampling corks are sent. When a RTT sampling cork is passed back to the TCP sender, the RTT is estimated by the difference between the current simulation time and the time when the cork was sent. Equation (4.21) can then be applied to compute the current retransmission timeout. However, recall that rate smoothing technique may smooth out the rate change with which a RTT sampling cork is associated. In order to prevent the complications involved in correcting such errors, a policy is added at every FIFO port that smoothing out any rate change that carries a RTT sampling cork is forbidden.

When the retransmission timer fires after $Timeout$, the mode transition threshold $ssthresh$ is set to be half of the congestion window size, and the congestion window size $cwnd$ is set to be one MSS. In order to retransmit the lost bytes, LBS is simply set equal to LBA.

As a loss clearance cork is returned to the TCP sender, δ_{loss} is calculated as Equation (4.20). If δ_{loss} is less than a MSS, then the TCP sender applies the fast retransmit logic: shrink the mode transition threshold $ssthresh$ to be half of the current congestion window size, set the congestion window size $cwnd$ to be $ssthresh$ plus 3 MSSes, and set the mode to be congestion avoidance. In order to retransmit the lost bytes, LBS is set equal to LBA.

4.4.3 Simulation Results

The dumbbell topology depicted in Figure 4.11 is used to study the accuracy and performance of the hybrid simulation of TCP traffic. Every link in the topology has the same bandwidth and propagation latency. Two sets of experiments are designed. In the first set of experiments, we use 10 TCP clients and 10 TCP servers. Each TCP client requests a file transfer of 5M bytes from its peer server; After a client receives all the requested bytes from the counterpart server, it keeps "silent" for an exponentially distributed period with mean of 5 seconds. In the second set of experiments, there are 100 TCP clients and 100 TCP servers; the traffic pattern of every client-server pair is the same as in the first set of experiments. In each set of experiments, we vary the link bandwidth between 5Mbps, 50Mbps and 500Mbps; we also vary the link propagation latency between 5 ms, 50 ms and 500 ms. Therefore, each set of experiments has 9 configurations. The simulation length of each configuration is 3600 seconds. We use the hybrid simulator and pure packet-level simulator to simulate each configuration independently.

In the hybrid simulation, we represent 20 percent of TCP flows with packet-oriented models and the other 80 percent of TCP flows with fluid-oriented models. Therefore, in the first set of experiments, 2 TCP flows are modeled as packet-oriented TCP and 8 TCP flows as fluid-oriented TCP; in the second set of experiments, there are 20 packet-oriented TCP flows and 80 fluid-oriented TCP flows. In the simulation, we are interested in packet-level TCP statistics like TCP goodput and round trip times. We also consider network statistics like average packet loss probability at the bottleneck port. The relative speedup of the hybrid simulation over the pure packet-level simulation is also studied because it offers us insights into the performance gain from the fluid-oriented TCP

model used in the hybrid simulation.

4.4.3.1 Accuracy

As we will see later from the simulation results, when there are many TCP flows traversing through the same bottleneck port, the behavior of each individual TCP flow is dynamic, causing the collected samples, except the round trip times, to exhibit high variation. Hence, it is difficult to quantify the accuracy of the statistics collected from the hybrid simulation using methods like relative errors. Instead, we focus on whether the results collected from the hybrid simulation are reasonable. If their mean falls within the range in which the samples are collected from the pure packet-level simulation, the results are viewed as totally reasonable; otherwise, the distance that it is from that range suggests how reasonable it is.

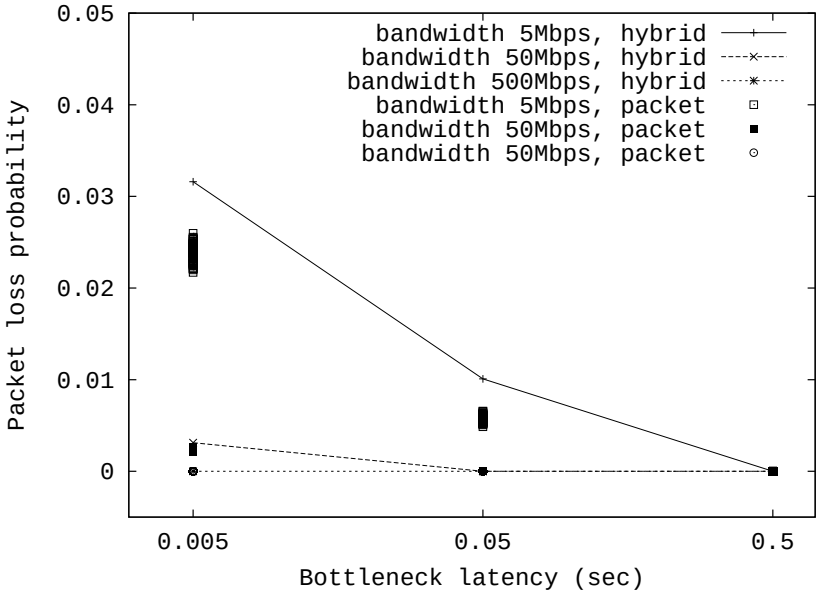


Figure 4.24: Packet Loss Probability under 10 TCP Flows

Statistics on average packet loss probabilities, TCP goodputs, and round trip times are described in Figures 4.24-4.29. From the simulation results, we have observed that in most cases the hybrid simulation and the pure packet-level simulation achieve excellent agreement on all collected statistics, but there is a visible discrepancy on average packet loss probability and TCP goodput when the bottleneck bandwidth is the lowest. The exception can be explained as follows. If the bottleneck bandwidth is higher, there is less mutual interaction between packet-oriented and fluid-oriented traffic at the bottleneck port, thus reducing the impact of any error caused by the model integrating hybrid traffic as discussed in Section 4.3. On the other hand, if the bottleneck link bandwidth is

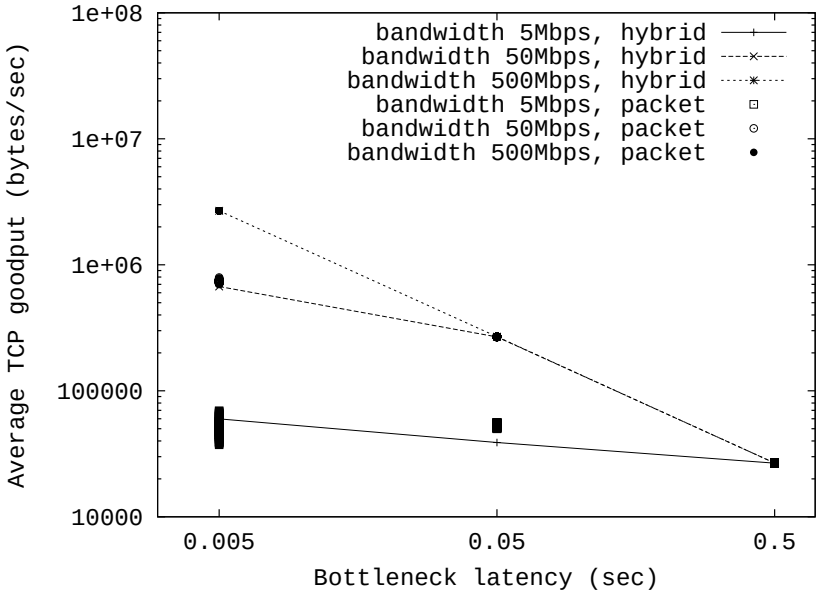


Figure 4.25: TCP Goodput under 10 TCP Flows

lower, the congestion level at the bottleneck port becomes higher and more bytes are thus lost from the fluid-based TCP streams. As discussed in Section 4.4.2, the fluid-based TCP model's response to lost traffic is a heuristics-based approximation to the real TCP's behavior. Increasing the congestion level thus increases the likelihood that higher errors result from that loss recovery model.

A closer examination reveals that the hybrid simulation overestimates the packet loss probability when the bottleneck port is under heavy TCP traffic load. For example, when the bottleneck latency is 5 milliseconds, the average packet loss probability in hybrid simulation is higher than the highest packet loss probability that an individual TCP stream suffers in the pure packet-level simulation, regardless of the number of TCP flows in the simulation. The phenomenon results from the early window reduction syndrome mentioned in Section 4.4.2. If the congestion level at the bottleneck port is very high, all the fluid TCP flows suffer either retransmission timeouts or fast retransmit after some congestion is detected at the bottleneck port. In either case, the congestion window sizes of some fluid TCP flows deflate earlier than they should. This allows some packet-oriented TCP flows to inflate their congestion windows up to a higher size than that in the pure packet-level simulation. Therefore, it is observed from the simulation results that in the region of low link bandwidth and low bottleneck latency, hybrid simulation produces higher average packet-level TCP goodputs than pure packet-level simulation. This seems to contradict the higher packet loss probabilities observed in hybrid simulation. Actually, because packet-oriented TCP flows are able to reach a higher congestion window size, the burstiness of their traffic results in more TCP packets being dropped at the bottleneck port in later rounds.

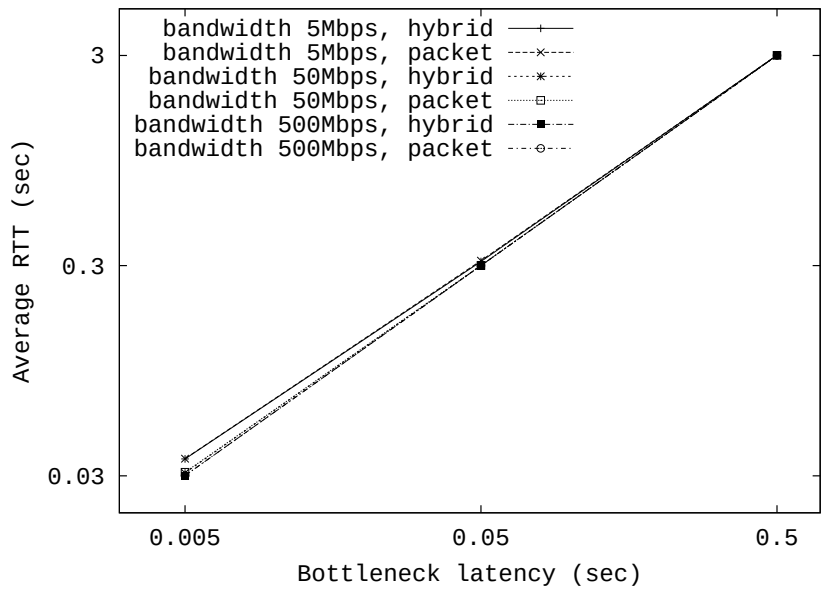


Figure 4.26: TCP Round Trip Time under 10 TCP Flows

Although the hybrid simulation overestimates the packet loss probabilities at the bottleneck port under heavy load, most of the TCP goodputs observed from the simulation results are feasible. In addition, the hybrid simulation also predicts well the average round trip times of TCP packets. Regardless of the number of TCP flows in the network, the relative error on this metric is below 4.0% under every network configuration.

In summary, the simulation results tell us that our hybrid model works fairly well under light and medium TCP traffic load in the network. But when the TCP traffic load in the network is high, the hybrid model is prone to overestimate the packet loss probability at the bottleneck port, but still produces feasible predictions on the average TCP goodputs and round trip times. However, for many applications whose behavior is sensitive to TCP statistics not network statistics, our hybrid model provides a good solution.

4.4.3.2 Execution Speedup

In this section, we discuss the relative execution speedup of the hybrid simulation over the pure packet-level simulation. Note that all the TCP flows are homogeneous. Hence, we can establish the upper bound on the execution speedup. Let T_{hybrid} be the total execution time of the hybrid simulation. It consists of three parts:

$$T_{hybrid} = T_{packet} + T_{fluid} + T_{interaction} \tag{4.22}$$

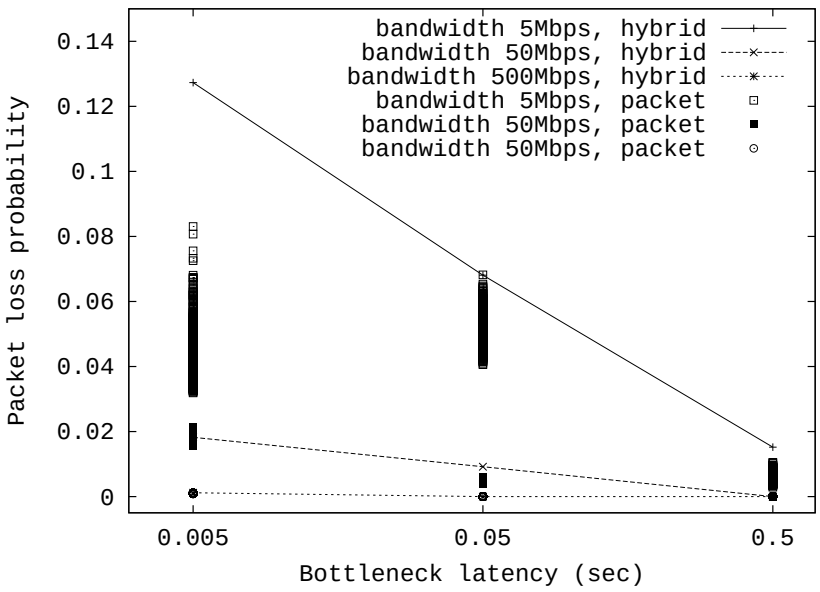


Figure 4.27: Packet Loss Probability under 100 TCP Flows

where T_{packet} is the execution time consumed on simulating packet-oriented TCP flows, T_{fluid} is the execution time consumed on fluid-oriented TCP flows, and $T_{interaction}$ is the execution time consumed on processing the interactions between packet-oriented and fluid-oriented TCP flows at the bottleneck port. In an extreme case where T_{fluid} and $T_{interaction}$ can be totally ignored, the execution time of the hybrid simulation is equivalent to that of simulating only the packet-oriented TCP flows. Since the proportion of the packet-oriented TCP flows is 20% in the hybrid simulation, its execution speedup over the pure packet-level simulation is bounded by 5 from the upper side.

Figure 4.30 describes the execution speedup of hybrid simulation over pure packet-level simulation for both sets of experiments. There are 3 curves that have execution speedups close to the upper bound throughout all three bottleneck latencies. They correspond to bottleneck bandwidth 500Mbps under both 10 and 100 TCP flows, and bottleneck bandwidth 50Mbps under 10 TCP flows. Common to them is relatively high bottleneck bandwidth with respect to the number of TCP flows traversing through the bottleneck port. Another 2 curves have low speedups when the bottleneck latency is the lowest, but their speedup is close to the upper bound when the bottleneck latency is 50 or 500 milliseconds. They correspond to bottleneck bandwidth 5Mbps under 10 TCP flows and bottleneck bandwidth 50Mbps under 100 TCP flows. Common to these two curves is medium bottleneck bandwidth available with respect to the number of TCP flows traversing through the bottleneck port. The last one, which corresponds to bottleneck bandwidth 5Mbps under 100 TCP flows, has low speedups when the bottleneck latency is 5 or 50 milliseconds, and its speedup increases up to about 3.5 when the bottleneck latency is 500 milliseconds.

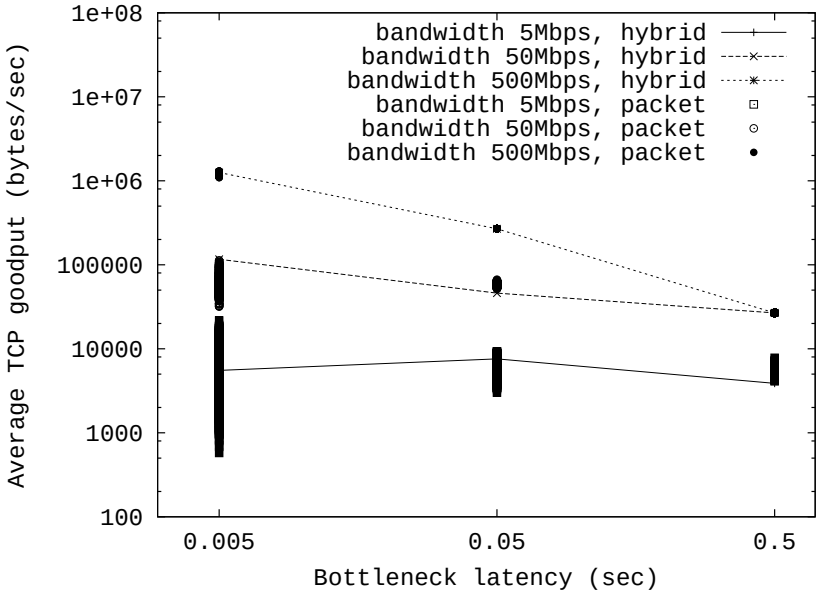


Figure 4.28: TCP Goodput under 100 TCP Flows

One trend observed from the simulation results is that a higher bottleneck bandwidth and a longer bottleneck latency are both helpful in achieving better relative speedup from the hybrid simulation. As we have seen in Figures 4.24 and 4.27, the congestion level at the bottleneck port increases as the bottleneck bandwidth or the bottleneck latency decreases. An increased congestion level causes the hybrid simulator to spend more computation time on processing fluid-oriented TCP traffic at the bottleneck port, but for the pure packet-level simulator, increasing the congestion level leads to more packets dropped at the bottleneck port and thus reduces the execution time. Therefore, both a higher bottleneck bandwidth and a longer bottleneck latency bring better execution speedup from the hybrid simulation over the pure packet-level simulation.

It has been noticed that the hybrid simulator is outperformed by the pure packet-level simulator when there are 100 TCP flows and links have bandwidth 5Mbps and latency 5 milliseconds. Under this configuration, simulating the network for one hour with the hybrid simulator requires 96 seconds, but that using the pure packet-level simulator requires only 76 seconds. This suggests that under extremely heavy traffic load, packet-oriented simulation may be a better choice from the performance perspective.

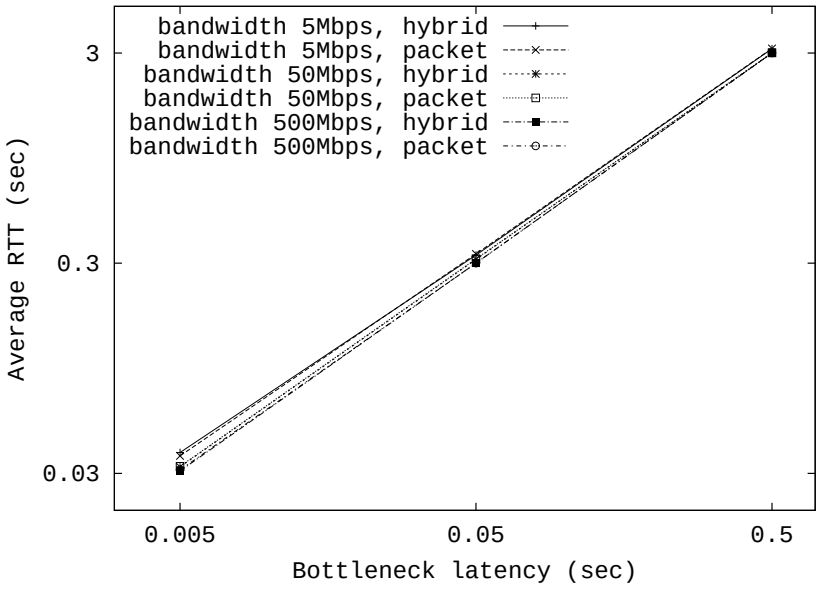


Figure 4.29: TCP Round Trip Time under 100 TCP Flows

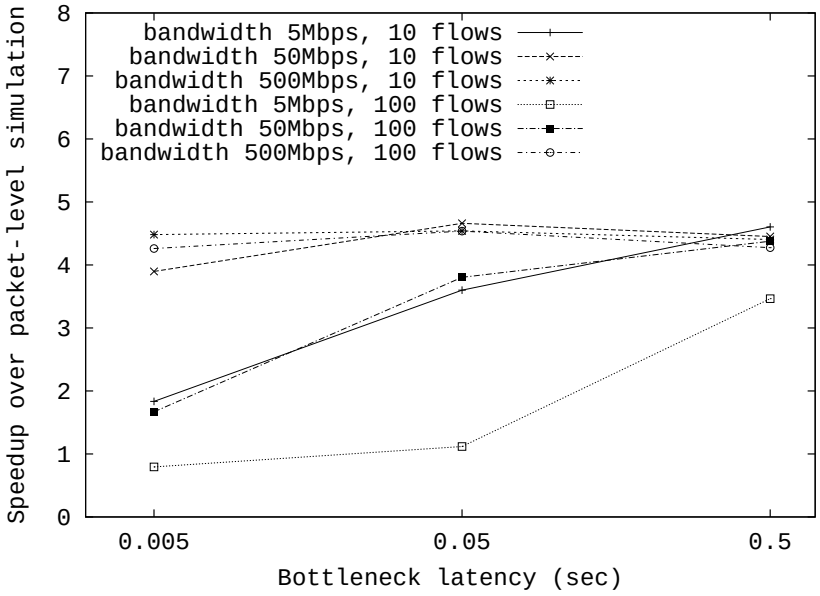


Figure 4.30: Execution Speedup of Hybrid Simulation over Pure Packet-Level Simulation

4.5 Related Work

The ripple effect associated with event-driven fluid simulation has been observed in early work on fluid simulation such as [61]. In [79], an approach to dampening ripple effect is described. In the method, flows in a network are combined into fewer aggregate flows. Hence, when the input rate of the aggregate flow changes, only one rate change is propagated to the downstream network component. The viability of this solution is contingent on two premises. On one hand, flows being aggregated must be destined for the same place; otherwise, when an aggregate flow splits into multiple destinations, we are unable to restore the rate change destined to a particular destination from the aggregate flow. On the other hand, the destination of each flow should not respond to the rate changes it receives in the simulation. Otherwise, if a fluid flow uses a responsive protocol like TCP, aggregating it into a larger flow in the network inevitably loses its detailed flow rate change information needed by the corresponding end application.

In [63], an alternative solution called *fluid threshold policy* is proposed to mitigate ripple effect. In this method, a new departure rate of a flow is propagated to the next network component only if it differs from the old departure rate by a percentage higher than a given threshold. In other words, if the two consecutive departure rate changes corresponding to the same flow are very close to each other, the one that occurs later in simulation time is ignored. This solution disobeys the flow conservation principle, which dictates that after a flow traverses through a lossless network, the destination should receive exactly the same amount of traffic as the source has sent out after a finite delay in simulation time. If the fluid threshold policy is applied as described, the destination may receive less or more than what the source has sent out. Hence, this method makes it difficult to implement some volume-aware applications like large file transfers unless the network provides the error information to the end applications. In addition, as pointed out by the author, errors derived from this method can propagate through the network. An error that happens at a port is embodied in the arrival rate at the next hop, where it may cause further errors due to the fluid threshold policy. In a network where a flow suffers errors at multiple ports on its path, the accumulated error at its receiver may be significant. Finally, in contrast to our rate smoothing technique, the fluid threshold policy lacks a strict upper bound on the number of rate changes that are received by network components in event-driven fluid simulation.

There are a few examples that integrate packet-oriented and event-driven fluid-oriented traffic simulation into the same network simulator. In [126], integration of fluid-oriented and packet-oriented network simulations is considered. The HDCF-NS (Hybrid Discrete-Continuous Flow Network Simulator) [93], which models behavior of fluid flows with discrete events, is used to generate fluid-oriented background traffic for the packet-oriented foreground traffic simulated with the *pdns* simulator[2]. The two simulators simulate the same topology simultaneously in separate address spaces, and the fluid-oriented background traffic simulation, as it evolves, updates the buffer occupancy in the packet-oriented foreground traffic simulator. In contrast to our approach, their method does not consider the impact that packet flows have on fluid flows. It thus eliminates the necessity of estimating flow rates for packet-oriented traffic, as is needed in our approach. The accuracy of their approach is contingent on the premise that the packet-oriented foreground traffic can be ignored

[2]http://www.cc.gatech.edu/computing/compass/pdns/index.html

when competing for bandwidth and buffer storage with fluid-oriented background traffic. If the packet-oriented foreground traffic is significant as opposed to the fluid-oriented background traffic, the communication bandwidth allocated to fluid flows will be overestimated.

The work in [64][63] bears some similarities to ours. The integration of fluid-oriented and packet-oriented traffic is done in the IP-TN simulator [130]. It also models the mutual interaction between fluid-oriented and packet-oriented traffic when they multiplex at the same port. In this method, a single state variable is used to keep the current buffer occupancy at each port, and this variable has dual purposes: determining the queueing delay when a packet arrives at the port and determining the fire time of fluid departure rate changes spawned by a fluid input rate change. This differs from our approach which uses two separate variables, one serving for each of the above purposes. Although our approach brings more complication to the implementation, it reduces the negative impact that the errors resulting from packet rate estimation impose on the calculation of buffer occupancy. In their approach, the packet arrival rate is estimated by summing the sizes of the last X packets and then dividing the sum by the time interval in which these packets arrive. Under extreme cases, the estimation approach may suffer significant inaccuracy. For example, if the last X packets arrive from X different input ports at the same time in the simulation, the time interval is 0 and the packet rate estimate is infinitely large. This conforms to the observation we have made from our own experiments: estimating packet rate is a delicate process and it may make the simulation results very sensitive to the parameter settings. In our approach, we use separate methods to estimate packet rates, one for competing bandwidth and the other for computing dropping probability, because from a lot of experiments we have done, it is observed that the accuracy of the simulation results is less sensitive to the former than the latter, especially when the fluid-oriented traffic is a dominant portion of the overall traffic.

4.6 Summary

In this chapter, we have discussed event-driven fluid-oriented traffic simulation and its integration with packet-oriented traffic simulation. There are some fluid-oriented traffic sources like MMFM and fluid-based TCP that generate fluid rate changes at discrete times. When fluid rate changes from more than one source multiplex at the same port in the network, they affect each other's departure rates if the port is overloaded. In a network where there exist multiple congested ports, "ripple effect" may occur in event-driven fluid-oriented traffic simulation, causing explosion of fluid rate changes in it. We have proposed a rate smoothing technique to mitigate "ripple effect". The key idea of this approach is that it exploits the insensitive latency of a port when fluid rate changes traverse through the associated link so that rate changes corresponding to the same flow within a short time interval can be flattened. We noticed that unconstrained rate smoothing may cause too much error to the simulation results, an adjustable time constraint on this technique offers a tradeoff between accuracy and efficiency.

We also have presented an approach to integrating fluid-oriented and packet-oriented traffic simulation in the same discrete event simulator. In our model, both the effect that packet-oriented traffic has on fluid-oriented traffic and the effect vice versa have been modeled. We have done

some experiments, using the MMFM model to generate fluid-oriented traffic. From the simulation results, close agreements have been observed between the hybrid simulation and the counterpart pure packet-oriented simulation. We have also seen execution speedups at varying orders from the hybrid simulation over the pure packet-oriented simulation.

Finally, we have studied the accuracy and performance of hybrid TCP traffic simulation. Nicol's fluid-based TCP model has been modified to generate fluid-oriented traffic bearing TCP characteristics. From the simulation results with the dumbbell topology, we conclude that under light and medium traffic load, the hybrid simulator is able to achieve both reasonable accuracy results and good execution speedups against the pure packet-level simulator, but under heavy traffic load, the hybrid simulator is prone to overestimate the packet loss probability and can possibly be outperformed by the pure packet-level simulator.

Chapter 5

Time-Stepped Coarse-Grained Traffic Simulation

In Chapter 4, we have seen that event-driven fluid-oriented traffic simulation is able to achieve significant execution speedup over pure packet-level traffic simulation. Such performance gain results from the capability of fluid-oriented representations to abstract packet-level details within certain time scales. As the time scale of interest becomes coarser, flow rate changes can further be aggregated in the time domain, making event-driven time advancement inefficient.

Motivated by this observation, a time-stepped technique is proposed in this chapter to simulate network traffic at coarse time scales. It periodically updates the demand discharged by each fluid-oriented traffic source. At every time step, traffic is assumed to converge to a steady state in the network within a relatively short time as opposed to the time scale being considered. This eliminates the necessity of using discrete events to propagate new rates through the network; instead, the aggregate traffic load on each link is directly computed with an optimized algorithm. In order to achieve further scalability, the algorithm is parallelized on a distributed memory multiprocessor. The PPBP traffic model, which exhibits self-similarity observed in various types of networks, is used to generate the traffic demands for each fluid-oriented traffic source.

The remainder of this chapter is organized as follows. Section 5.1 discusses the motivation behind the time-stepped coarse-grain traffic simulation. In Section 5.2, the outline of our solution is provided. In Section 5.3, we formulate the problem to be investigated. In Section 5.5, a sequential algorithm is presented, followed by a discussion on its convergence behavior, performance and accuracy. In Section 5.7, we describe how to parallelize the sequential algorithm on a distribued memory multiprocessor; simulation results on the scalability of the parallel algorithm with both fixed and scaled problem size are also presented. Section 5.8 discusses how to modify the algorithm in a network that implements the GPS service discipline [113]. In Section 5.9, some related work is discussed. Section 5.10 summarizes this chapter.

5.1 Background

The motivating application for time-stepped coarse-grained traffic simulation is the RINSE (Real-Time Immersive Network Simulation Environment) project, which provides a real-time network simulation platform for cyber-security exercises and training [75]. Owing to human participation in RINSE, the underlying network simulator must run as fast as real time. Therefore, it poses a significant performance challenge to the simulator. The traffic in a network simulation can break down into two parts. Foreground traffic is the part whose behavior is of essential interest to the user, and background traffic is the part that itself is not concerned by the user but may affect the behavior of foreground traffic. For example, when RINSE is used to study how malicious worm traffic affects the ongoing financial transaction services, the transaction traffic is treated as foreground traffic.

In network simulation, background traffic usually constitutes a large fraction of the network traffic, or in many cases most of the network traffic. Hence, it is not a trivial problem to simulate background traffic efficiently in a real-time simulation environment like RINSE. The simplest way of simulating background traffic is to use a random traffic generator to generate the background traffic load on each link. This approach, although efficient and scalable, loses spatial correlation among background traffic on different links.

The (M, P, S) model [89] is a WAN (Wide Area Network) background traffic generation model. In its first phase, the FGN (Fractional Gaussian Noise) model [8], a self-similar traffic model, is used to generate a sequence of aggregated traffic traces from every traffic source in the network. Such a sequence is called an *aggregate stream*. In the second phase, an aggregate stream is decomposed into individual substreams, one for each destination campus network access point, by sampling some probability models. In the final phase, the packet-train model discussed in Section 2.2.2 is applied to convert each of these substreams to short-term packet arrivals. Although the (M, P, S) model achieves execution time savings because of the aggregated traffic stream per source campus network, the large number of packets generated after the final phase can make it difficult to satisfy the real time constraint.

An alternate way of simulating background traffic is using fluid-oriented models. Since TCP traffic dominates the current Internet traffic, fluid-based TCP models can be exploited to improve background traffic generation as observed in Chapter 4. However, modeling traffic in a large network at session level is still an arduous undertaking because of the enormous TCP sessions in it. In [85], a fluid-based TCP model using stochastic differential equations, combined with a time-stepped time advancement mechanism, is applied to simulate a ring topology with 1,045 nodes. The experimental results show that simulating 176,000 TCP flows in this network for 100 seconds takes about 74 minutes, which is far from real time.

5.2 Solution Outline

Since existing background generation methods are unable to satisfy the real time constraint required in RINSE, a time-stepped coarse-grained traffic simulation technique is developed to achieve this objective. Aggregation, a strategy often adopted to simplify simulation models (See Section 2.2.1),

is the key idea. In the space domain, all flows that traverse between the same source-destination pair are aggregated and treated as a single aggregate flow in the simulation. In a large network, usually many sessions between an ingress-egress pair use the same path. Under such circumstance, the flow aggregation technique can significantly reduce the number of flow states and thus the computation complexity. On the other hand, aggregation is also applied in the time domain. Simulating large-scale background traffic at fine time scales is computationally costly, but is sometimes unnecessary with regard to the simulation objective. Our approach, driven by constant time steps, simulates the network traffic at coarse time scales. At every time step, a new traffic matrix that contains traffic demands among all ingress-egress pairs is generated. The traffic demand between an ingress-egress pair is assumed to be unaltered between any two successive time steps. The transient behavior that occurs before the whole network traffic converges to a steady state is ignored because of the coarse time scale considered. Hence, this technique numerically computes the background traffic load on each link given the current traffic matrix, and this load remains unchanged until the next time step.

The relationship between foreground traffic simulation and background traffic computation is illustrated in Figure 5.1. Background traffic can affect how foreground traffic behaves. As mentioned before, after the background traffic computation at a time step, a background traffic load is calculated on every link in the network. It is this load that imposes impact on the foreground traffic traversing on the same link. Actually, from the perspective of foreground traffic, the aggregate background traffic traversing through a port can be viewed as a virtual event-driven fluid-oriented flow that immediately disappears into a virtual sink after departing from the port. Moreover, this virtual flow periodically updates its arrival rate as the background traffic load at this port computed at every time step. This virtual flow competes for bandwidth and buffer space with foreground traffic as if it were a regular event-driven fluid-oriented flow. In this way, background traffic can shape the behavior of foreground traffic.

On the other hand, foreground traffic can also affect background traffic. This conforms to the principle that traffic behavior should be transparent to the way in which traffic is represented in the simulation. If foreground traffic is as intense as background traffic, ignoring its impact on background traffic may introduce significant inaccuracy to the simulation results. In our approach, when the background traffic load on a link is computed, the volume of foreground traffic that it has "seen" since the previous time step is collected. Moreover, it is assumed that between the current time step and the next time step, the same amount of foreground traffic will traverse on this link. This simple model may not predict the real foreground traffic closely. It, however, can be easily replaced with a more powerful but more complicated prediction model like ARIMA (Autoregressive Integrated Moving Average) traffic model [10]. When we compute the background traffic load at an output port, it is assumed that there exists a virtual aggregate flow, which comes from the router associated with this port, crosses the link, and then immediately disappears into a sink; at every time step, this virtual aggregate flow injects an ingress rate that is equal to the total volume of foreground traffic seen since the previous time step divided by the time step size. Therefore, our approach also captures the impact of foreground traffic on background traffic in the simulation.

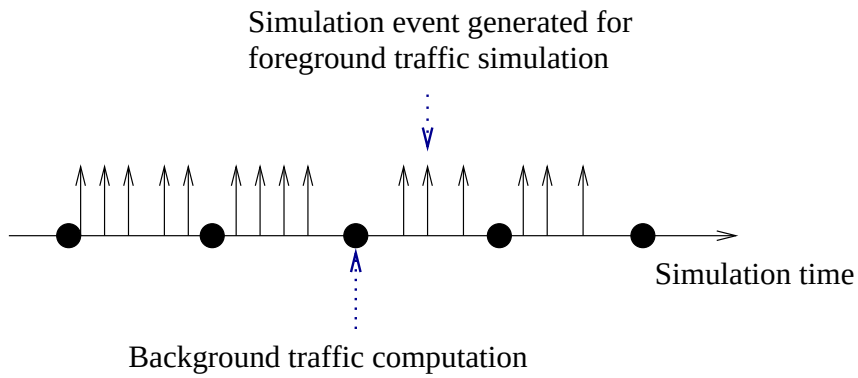


Figure 5.1: Relationship between Foreground Traffic Simulation and Background Traffic Computation

5.3 Problem Formulation

In the time-stepped traffic simulation, the simulation time is discretized into units of length Δ, and we use t_k to denote $k \cdot \Delta$ ($k = 0, 1, 2, ...$). We assume that the network being studied consists of n POPs, denoted by P_1, P_2, ..., P_n. We use $f_{i,j}$ to represent the aggregate flow between ingress-egress pair $< P_i, P_j >$. For any ingress-egress pair $< P_i, P_j >$ ($1 \leq i, j \leq n$), we use $T_{i,j}(t)$ to denote ingress rate of the corresponding aggregate flow at time t. Since the simulation time advances by constant time intervals, we need to discretize the ingress rate for every aggregate flow. During the time interval $[t_k, t_{k+1}]$, the traffic volume emitted from ingress node P_i to egress node P_j is

$$\int_{t_k}^{t_{k+1}} T_{i,j}(t)\mathrm{d}t. \tag{5.1}$$

We smooth the burstiness at time scales smaller than Δ. In order to ensure that the same amount of traffic is injected into the network, the ingress rate of aggregate flow $f_{i,j}$ at discretized time $k \cdot \Delta$ is

$$\frac{1}{\Delta} \times \int_{t_k}^{t_{k+1}} T_{i,j}(t)\mathrm{d}t. \tag{5.2}$$

The network is modeled as a collection of routers connected with uni-directional links. The sending endpoint of a link is associated with a router's output port. At an output port, there is an output buffer. We assume that every router adopts the output buffering strategy, although we understand that other buffering strategies can be easily incorporated with minor modifications. Routing protocols are used to direct traffic for each aggregate flow. The routing decisions are assumed to be static within any discretized time interval. In addition, the time step size Δ is relatively large with respect to the typical end-to-end latency of an aggregate flow. Therefore, all link latencies are assumed to be zero in the network model.

Let set Q denote the whole set of output ports in the network. The sequence of output ports an aggregate flow $f_{i,j}$ traverses at discretized time t_k is denoted by $Q_{f_{i,j}}(t_k)$. Obviously, $Q_{f_{i,j}}(t_k)$ is a subset of Q. $F_q(t_k)$ is used to denote the whole set of aggregate flows that traverse port q ($q \in Q$) at discretized time t_k. We use $\lambda_{f,q}^{(in)}(t_k)$ ($f \in F_q(t_k)$) to denote the arrival rate of aggregate flow f at output port q at discretized time t_k. The sum of all the arrival rates into port q, $\sum_{f \in F_q(t_k)} \lambda_{f,q}^{(in)}(t_k)$, is written as $\Lambda_q^{(in)}(t_k)$. In addition, we use $\widehat{\lambda}_f(t_k)$ to denote the ingress rate of aggregate flow f from its traffic source at discretized time t_k.

When multiple aggregate flows traverse the same output port, the bandwidth is allocated to each is governed by the port's queueing policy. Consider an output port q with the FIFO queueing policy. Let μ_q denote the link bandwidth associated with q. According to the FIFO service discipline, the departure rate of flow f ($f \in F_q(t_k)$) at discretized time t_k, denoted by $\lambda_{f,q}^{(out)}(t_k)$, is as follows:

$$\lambda_{f,q}^{(out)}(t_k) = \lambda_{f,q}^{(in)}(t_k) \times \min\{1, \frac{\mu_q}{\Lambda_q^{(in)}(t_k)}\}. \tag{5.3}$$

When the aggregate arrival rate does not exceed the link bandwidth, the output port can serve all input traffic; otherwise, congestion occurs and traffic that cannot be served has to be dropped; the FIFO queueing policy dictates that the loss that occurring to each flow be proportional to its arrival rate. In Equation (5.3), we do not model the queueing delay, and therefore, there is no time shift between the departure rate and the arrival rate. This is a reasonable assumption because a coarse time scale is being considered.

As discussed in Section 5.2, one objective is to compute the aggregate traffic load at each port:

$$\sum_{f \in F_q(t_k)} \lambda_{f,q}^{(in)}(t_k) \quad \text{for every } q : q \in Q \tag{5.4}$$

For some applications, the egress rate of each aggregate flow needs to be reported to its sink. For example, the emission rate of an aggregate flow at a time step may be a function of the rate at which the ingress node receives traffic from the network at the previous time step. Hence, another objective is to compute the aggregate egress rate from each POP. In order to formulate this objective, we introduce another notation. We define function $\Pi(f, q)$, where $f \in F_q(t_k)$, as follows:

$$\Pi(f,q) = \begin{cases} q' & \text{if } q' \ (q' \in Q) \text{ is the next output port on flow } f\text{'s path after leaving port } q \\ P & \text{if flow } f \text{ arrives at its destination } P \ (P \in \{P_i\}_{1 \le i \le n}) \text{ after leaving port } q \end{cases} \tag{5.5}$$

We formulate the second objective as computing

$$\sum_{q \in Q, f \in F_q(t_k), \Pi(f,q)=P} \lambda_{f,q}^{(out)}(t_k), \quad \text{for every } P\text{: } P \in \{P_i\}_{1 \le i \le n}. \tag{5.6}$$

5.4 Solutions in Specific Settings

Given the problem formulated in Section 5.3, we introduce three types of networks and discuss how to solve the problem on these networks. Some examples are presented to help us understand the characteristics of the problem under varied settings.

5.4.1 Networks with Sufficient Bandwidth

We define a network with *sufficient bandwidth* as one in which the bandwidth associated with any link is no less than the sum of the ingress rates of all the flows that traverse on it. It is worthy of noting that a network with sufficient bandwidth does not necessarily have high-bandwidth links; a low-bandwidth network, if little traffic traverses through it, can still be deemed as a network with sufficient bandwidth. Networks with sufficient bandwidth are not uncommon in the Internet. In order to avoid deploying complicated QoS mechanisms, many network operators prefer to over-provision link bandwidth in their networks. This renders it possible that congestion seldom occurs in such networks. Moreover, after a new network is deployed, its traffic load is usually very low from the beginning. Such a network at its initial operational stage can also be deemed to have sufficient bandwidth. It is thus meaningful to discuss how to solve the problem formulated in Section 5.3 efficiently when the traffic load is so low that no congestion occurs in the network.

In a network with sufficient bandwidth, the solution can actually be very simple. Because no congestion should happen in the network, every flow is able to push all its traffic towards its destination without any loss. The aggregate traffic load at each output port can be simply calculated by adding the ingress rates of all the flows that traverse it. Hence, the aggregate traffic load at each port can be computed as

$$\sum_{f \in F_q(t_k)} \widehat{\lambda}_f(t_k) \text{ for every } q : q \in Q. \tag{5.7}$$

Similarly, the egress rate at a POP can be computed by adding the ingress rates of all the flows that are destined to it. At POP P ($P \in \{P_i\}_{1 \le i \le n}$), its aggregate egress rate is

$$\sum_{q \in Q, f \in F_q(t_k), \Pi(f,q) = P} \widehat{\lambda}_f(t_k). \tag{5.8}$$

For each flow that traverses in the network, the above solution reads or updates the state of every port on its path for a constant number of times. If we use N and M to denote the total number of flows in the network and the average number of ports on a flow's path, then the time complexity is simply $\Theta(N \cdot M)$.

5.4.2 Feed-Forward Networks with Limited Bandwidth

When the links in a network have limited bandwidth, it is likely that some of them are congested because the limited bandwidth cannot serve all the input traffic. The topology discussed in this

example is the feed-forward network shown in Figure 4.4. As we mentioned in Section 4.2.2, no cycle is formed in this network.

After the setup of every time step, we know the ingress rate from each flow's source. It is observed that at port A, the arrival rates of all the flows that traverse it are already known. Therefore, the departure rate of flow f_i ($1 \leq i \leq 4$) from port A can be determined using Equation (5.3). We say that we *resolve* a port if we use Equation (5.3) to compute the departure rates of all the flows traversing through it after all their arrival rates are known. After port A is resolved, the arrival rates of flows f_1 and f_2 into port B are known, and the arrival rates of flows f_3 and f_4 into port C are also known. Thereafter, we can further apply Equation (5.3) on ports B and C in an arbitrary order. Once port B is resolved, we know the arrival rates of flows f_5 and f_6 into port D, and the arrival rates of flows f_1 and f_2 into port E; then, we can resolve ports D and E in an arbitrary order. Once port C is resolved, we know the arrival rates of flows f_3 and f_4 into port D, and the arrival rates of flows f_7 and f_8 into port E; then, we can resolve ports F and G in an arbitrary order. After any of ports D, E, F, and G has been resolved, we can determine the departure rates of all the flows traversing through it.

A generalized algorithm for solving the problem formulated in Section 5.3 in feed-forward networks is described as follows. List L is used to maintain a collection of output ports. For every flow in the network, we put the first output port on its path onto list L. We then iterate over the ports on the list. In each iteration, we grab an arbitrary output port from L, resolve it, and settle the arrival rate of each traversing flow at the next output port on its path if it does not disappear into a sink; if we find an unresolved port to which the arrival rates of all the input flows are known, it is added onto L. Iterations terminate when L becomes empty. In this algorithm, the state of every output port is read or updated for a constant number of times and so does the flow variable associated with each of its input flows. Therefore, the time complexity of this algorithm is $\Theta(N \cdot M)$, where N and M denote the total number of flows in the network and the average number of output ports on a flow's path.

5.4.3 Networks with Circular Dependencies

When flows form circular dependencies, the solution to the problem formulated in Section 5.3 becomes more complicated. In the five-port topology shown in Figure 5.2 has 5 ports, A, B, C, D, and E. There are are 5 flows, f_1, f_2, f_3, f_4, and f_5, each traversing throughput 2 ports. We say $p \rightarrow q$ if the resolution of port q depends on that of port p. We then have: $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow A$. Hence, resolution of the 5 ports is circularly dependent, making the solutions discussed in Section 5.4.1 and Section 5.4.2 inapplicable here.

Following the notations in Section 5.3, we use $\lambda_{f,q}^{(in)}(t_k)$ to denote the arrival rate of flow f into port q at discretized time t_k and μ_q to denote the link bandwidth associated with port q. Using Equation (5.3), we form a group of equations as follows:

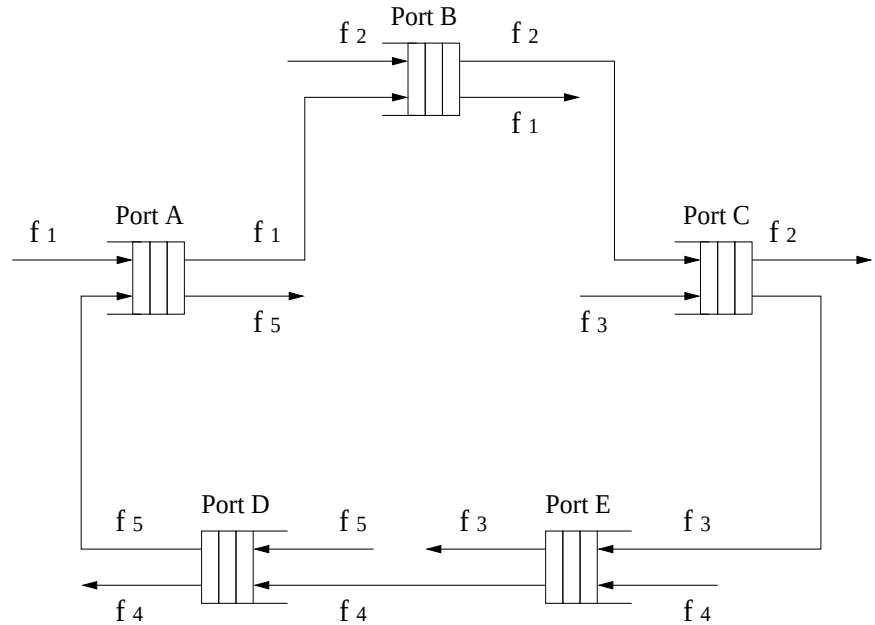


Figure 5.2: A Five-Port Ring Topology

$$\begin{cases} \lambda^{(in)}_{f_1,B} &= \lambda^{(in)}_{f_1,A} \times \min\{1, \mu_A/(\lambda^{(in)}_{f_1,A} + \lambda^{(in)}_{f_5,A})\} \\ \lambda^{(in)}_{f_2,C} &= \lambda^{(in)}_{f_2,B} \times \min\{1, \mu_B/(\lambda^{(in)}_{f_2,B} + \lambda^{(in)}_{f_1,B})\} \\ \lambda^{(in)}_{f_3,D} &= \lambda^{(in)}_{f_3,C} \times \min\{1, \mu_C/(\lambda^{(in)}_{f_3,C} + \lambda^{(in)}_{f_1,C})\} \\ \lambda^{(in)}_{f_4,E} &= \lambda^{(in)}_{f_4,D} \times \min\{1, \mu_D/(\lambda^{(in)}_{f_4,D} + \lambda^{(in)}_{f_1,D})\} \\ \lambda^{(in)}_{f_5,A} &= \lambda^{(in)}_{f_5,E} \times \min\{1, \mu_E/(\lambda^{(in)}_{f_5,E} + \lambda^{(in)}_{f_1,E})\} \end{cases} \tag{5.9}$$

where $\lambda^{(in)}_{f_1,B}$, $\lambda^{(in)}_{f_2,C}$, $\lambda^{(in)}_{f_3,D}$, $\lambda^{(in)}_{f_4,E}$, and $\lambda^{(in)}_{f_5,A}$ are flow variables whose values are to be determined. Equation (5.9) is actually a system of non-linear equations, because of the components on the right side that shape traffic when congestion occurs. In a generalized network, it can be abstracted into

$$H(\vec{\boldsymbol{\lambda}}) = \vec{\boldsymbol{\lambda}}, \tag{5.10}$$

where $\vec{\boldsymbol{\lambda}}$ is a vector of d variables, λ_1, λ_2, ..., λ_d, and $H(\cdot)$ is a d-dimension vector-valued function. Each variable in vector $\vec{\boldsymbol{\lambda}}$ corresponds to the arrival rate of a flow at an output port. In addition, for every output port on a flow's path, there is a variable in vector $\vec{\boldsymbol{\lambda}}$ that corresponds to the flow's arrival rate into that port.

Equation (5.10) suggests that its root is actually a fixed point of function $H(\cdot)$. This opens the avenue of applying fixed point iterations to solve the original problem. More specifically, an iterative algorithm works as follows:

(1) Pick an initial estimate $\vec{\lambda}^{[0]}$;

(2) For $k = 0, 1, ...,$, $\vec{\lambda}^{[k+1]} = H(\vec{\lambda}^{[k]})$.

(3) The iteration in (2) terminates as some norm of distance between $\vec{\lambda}^{[k]}$ and $\vec{\lambda}^{[k+1]}$ is closer than a tolerance ϵ.

We carefully implement step (2) to ensure that in each iteration every flow variable in $\vec{\lambda}$ is read or updated for only a constant number of times. We use P_f to denote the whole set of ports that have at least one input flow. Then we have $|\vec{\lambda}| \geq |P_f|$. Consider the iteration when k is k_z. Given the current estimate $\vec{\lambda}^{[k_z]}$, we compute the sum of all the input flow rates at each port in P_f. Hence, every flow variable in $\vec{\lambda}$ is read once so far. After the aggregate input rate into a port is known, we apply Equation (5.3) to compute the departure rate of each flow that traverses the port and then use it to update the estimate on the input flow rate at the next hop for the next iteration. Each input flow variable in $\vec{\lambda}$ is read or updated for only a constant number of times in the implementation.

We call the algorithm implemented as discribed the *fixed point algorithm*. We still use N and M to denote the total number of flows in the network and the average number of ports on a flow's path respectively. The number of flow variables in $\vec{\lambda}$ is thus $N \cdot M$. Then we can establish the following lemma:

Lemma 2 *The time complexity of each fixed point iteration in the fixed point algorithm is* $\Theta(N \cdot M)$.

In addition to the computation cost on each iteration, the performance of the fixed point algorithm is contingent on how many iterations are done before it terminates. There are some important issues regarding the iterative solution to Equation (5.10). They include whether the algorithm converges, whether there exists a unique fixed point solution, and how fast the algorithm converges. The behavior of fixed point iterations can vary widely: it may converge rapidly, converge slowly, or even diverge, depending on the problem under analysis. We will continue this discussion in Section 5.5.2. For the moment, we assume that the fixed point algorithm terminates after a finite number of iterations.

The simplest estimate on the root of Equation (5.10) is that all variables in $\vec{\lambda}^{[0]}$ are 0, that is, $\vec{\lambda}^{[0]} = \vec{0}$ where $\vec{0}$ is a d-dimension vector of all zeros. In other words, the initial network state is assumed to be empty: no flow has injected any traffic into the interior network except at its ingress port. The fixed point iterations, then, gradually propagate every flow's ingress rate towards its destination. When Equation (5.3) is applied on some flow at a port, its aggregate input rate may be 0. We let $0/0$ be 1 in the computation when that occurs.

If the fixed point algorithm takes K iterations to find a fixed point, Lemma 2 tells us that its time complexity is $\Theta(K \cdot N \cdot M)$. The fixed point algorithm can be applied on any type of networks

to solve the problem formulated in Section 5.3. Since the fixed point algorithm provides a general solution to the problem, how does it perform in the specific networks discussed in Section 5.4.1 and Section 5.4.2? The following lemmas answer this question.

Lemma 3 *Given a network with sufficient bandwidth, if the longest flow path has l ports, then the fixed point algorithm with initial estimate $\vec{\mathbf{0}}$ finishes after l iterations.*

Proof: In each iteration in the fixed point algorithm, the ingress rate of any flow is propagated one hop further towards its destination because the link bandwidth associated with any port is sufficient to serve all its input traffic. Provided that the longest path of a flow is l, the ingress rate of any flow must have been propagated to its sink after $l - 1$ iterations. At this time, every variable in $\vec{\boldsymbol{\lambda}}$ must have been set to be the ingress rate of the corresponding flow. An extra iteration is needed to ensure that the termination criterion can be satisfied. Therefore, the fixed point algorithm terminates after l iterations. □

From Lemma 3, we can obtain the following corollary:

Corollary 3 *Given a network with sufficient bandwidth, if the longest flow path has l ports, the time complexity of the fixed point algorithm is $\Theta(l \cdot N \cdot M)$.*

Recall that the solution in Section 5.4.1 has asymptotic time complexity $\Theta(N \cdot M)$. Corollary 3 tells us that the fixed point algorithm, if directly applied on networks with sufficient bandwidth, does not have the optimal performance.

Given a network topology and the flows that traverse it, we construct a graph whose vertices are the output ports in the network: if there exists a flow that traverses output ports q_1 and q_2 consecutively, then there is a directed edge from q_1 to q_2 in the graph. We say the graph constructed as described is the *condensed graph* of the original topology. We define the *diameter* of the condensed graph to be the length of the longest directed path in the graph on which no output port appears more than once. For example, the condensed graph of the feed-forward network shown in Figure 4.4 is illustrated in Figure 5.3.

Lemma 4 *Given a feed-forward network and the flows that traverse it, if its condensed graph has diameter ω, the fixed point algorithm with initial estimate $\vec{\mathbf{0}}$ finishes after at most ω iterations.*

Proof: In the condensed graph, we define the depth of a port q, denoted by γ_q, as the number of ports on the longest path from any other port to port q by following directed edges in the graph. Given a feed-forward network, there must exist at least one port whose depth is 1, and no port appears on any path more than one times.

We claim that the arrival rates to every port whose depth is no greater than j do not change after $j - 1$ iterations, where $j = 1, 2, \ldots$. We prove it by induction. When j is 1, any port whose depth is 1 must be the ingress port for all the flows traversing through it. Obviously, the ingress rate of a flow does not change in later iterations. We assume the arrival rates to every port whose depth is no greater than i do not change after $i - 1$ iterations. Consider any port, say port q, whose depth is $i + 1$. We use $prev(q)$ to denote the whole set of ports that have a directed edge pointing to port q in

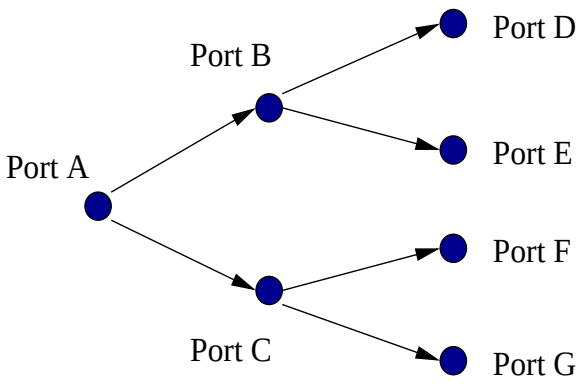


Figure 5.3: The Condensed Graph of The Feed-Forward Network Shown in Figure 4.4

the graph. We claim that for any port in $prev(q)$, its depth must be no greater than i. We prove it by contradiction. Suppose the depth of port q' in $prev(q)$ is greater than i. According to the definition of a port's depth in the condensed graph, there must exist a path that has more than i ports and also has port q' as its endpoint. In a feed-forward network, port q must not appear on this path because there exists an edge from port q' to port q. Then, a new path can be constructed by combining the path to port q' and the edge from port q' to port q. This new path has more than $i+1$ ports on it and the depth of port q must be greater than $i+1$. This contradicts the assumption that the depth of port q is $i+1$. Therefore, for any port in $prev(q)$, its depth must be no greater than i. Based on the assumption that arrival rates to every port whose depth is no greater than i do not change after $i-1$ iterations, we know that arrival rates to port q should not change after i iterations. By induction, arrival rates to every port whose depth is no greater than j must not change after $j-1$ iterations, where $j = 1, 2, ...$

If the diameter of the condensed graph is ω, then the largest depth that a port can have is at most ω. Therefore, no arrival rates change after $\omega - 1$ iterations based on the above claim. An extra iteration is used to ensure that the termination criterion can be satisfied. Therefore, the fixed point algorithm terminates after at most ω iterations. □

We have the following corollary regarding the time complexity of the fixed point algorithm when it is applied on a feed-forward network:

Corollary 4 *Given a feed-forward network and the flows that traverse it, if the directed graph constructed as described has diameter ω and the longest flow path has l ports, the time complexity of the fixed point algorithm is $O(\omega \cdot N \cdot M)$ and $\Omega(l \cdot N \cdot M)$.*

Proof: From Lemma 4, we know that the time complexity of the fixed point algorithm is $O(\omega \cdot N \cdot M)$. On the other hand, because the longest flow path has l ports, at least $l-1$ iterations are needed to ensure that the ingress rate of each flow has been propagated to every port on its path. Therefore, the time complexity of the fixed point algorithm is $\Omega(l \cdot N \cdot M)$. □

Recall that the list-based solution discussed in Section 5.4.2 has asymptotic time complexity $\Theta(N \cdot M)$. Corollary 4 tells us that the fixed point algorithm is not an optimal solution when the network being considered is a feed-forward network.

If there exists circular dependency in the network as shown in Figure 5.2, the fixed point algorithm is able to find the solution. Under some circumstance, however, the upper bound information on the arrival rates can be exploited to break the circular dependence among flow variables, making fixed point iterations unnecessary. Use the five-port ring topology shown in Figure 5.2 as an example. Suppose that the ingress rate of every flow is 1Mbps. The link bandwidth associated with port A is 2Mbps, and the link bandwidth associated with any other port is 1Mbps. Although we cannot compute the departure rate of either of f_4 and f_5 from port D, we can establish the upper bound on the departure rate of f_5 from there. From Equation (5.3), the departure rate of f_5 should be no greater than the arrival rate of f_5 into port D, which is 1Mbps. If this information is known to port A, the upper bound on the arrival rate of flow f_5 into port A can also be established. Now we know that the aggregate arrival rate into port A is at most 2Mbps, which means that all input traffic from flows f_1 and f_5 can be served without any loss. Therefore, the departure rate of flow f_1 from port A must be 1Mbps, and so is its arrival rate into port B. With knowledge of the arrival rates of both flows f_1 and f_2 into port B, we can derive their departure rates from port B as both (1/2) Mbps using Equation (5.3). We then know the arrival rate of flow f_2 into port C to be (1/2) Mbps, and we can thus derive the departure rates of flows f_2 and f_3 from port C as (1/3) Mbps and (2/3) Mbps respectively. Similarly, we can derive that the arrival rate of flow f_3 into port E is (2/3) Mbps, the arrival rate of flow f_4 into port D is (3/5) Mbps, and the arrival rate of flow f_5 into port A is (5/8) Mbps.

The above example suggests that even though circular dependency may exist among the flow variables in some networks, their upper bound information can be exploited to identify some ports whose aggregate arrival rates are no greater than the link bandwidths associated with them. We call such ports *transparent*. Once a transparent port is identified, all the arrival rates that have already been determined can safely be propagated to the downstream ports because no congestion occurs to this transparent port. These settled rates may render some downstream port resolvable and may even break the whole circular dependency in the network, as we have seen in the above example.

The idea of aggressively identifying transparent ports by setting tight upper bounds on flow variables may not remove all the circular dependencies among unsettled flow variables. In order to determine their values under such circumstance, the fixed point algorithm is still necessary. However, this strategy minimizes the number of unsettled flow variables involved in the fixed point algorithm. As we have analyzed above, the performance of the fixed point algorithm is contingent on both the number of iterations executed before the termination criterion is satisfied and the number of flow variables that are involved in each iteration. Hence, if many variables in $\vec{\boldsymbol{\lambda}}$ are determined before the fixed point algorithm is applied, the computation cost on the fixed point iterations can be significantly reduced.

5.5 Sequential Algorithm

In the previous section, we have observed that some algorithms have relatively good performance on specific networks like feed-forward networks and ones that have sufficient bandwidth, but are unsuitable for the networks that have circular dependencies. On the other hand, the fixed point algorithm works on any type of networks, including ones that involve circular dependencies, but suffers poor performance on some specific networks. A natural question following the above observations is:

Question 4 *Is there any solution to the problem formulated in Section 5.3 that not only provides good performance on the aforementioned specific networks but also solve the problem on networks involving circular dependencies?*

This section presents such a solution and thereafter discusses its convergence behavior, performance, and accuracy. We are limited to sequential computing architectures here and leave its parallelization in Section 5.7.

5.5.1 Algorithm Description

We use the same notations in the previous sections. A state variable is associated with each flow variable in $\vec{\boldsymbol{\lambda}}$. $S(\lambda_i)$ is used to denote the state variable associated with flow variable λ_i, where $1 \leq i \leq |\vec{\boldsymbol{\lambda}}|$. There are three possible states for state variable $S(\lambda_i)$: *unsettled*, *bounded*, and *settled*. The meanings of these states are explained as follows:

- for a flow variable in an *unsettled* state, we know neither its exact rate nor its upper bound;
- for a flow variable in a *bounded* state, we do not know its exact rate but know its upper bound;
- for a flow variable in a *settled* state, we know its exact rate.

As introduced in Section 5.2, a new ingress rate is generated for every flow in the network at the beginning of every time step. Then, for every flow variable in $\vec{\boldsymbol{\lambda}}$ that corresponds to the arrival rate of a flow at its ingress port, we set its state to be *settled* and its value to be the ingress rate; for any other variable in $\vec{\boldsymbol{\lambda}}$, we set its state to be *unsettled* and its value to be 0. Throughout the algorithm, the state transition of a flow variable can only be from *unsettled* to *bounded*, from *unsettled* to *settled*, or from *bounded* to *settled*. This is illustrated in Figure 5.4. Hence, a state transition of a flow variable always means that more knowledge on it has been gained. In addition, we say that we *settle* a flow variable if we determine the rate of that variable and change its state to *settled*.

We also associate a state variable with every output port in the network. Without introducing confusion, we use $S(q)$ to denote the state variable of output port q. Three possible states for a port are: *unresolved*, *transparent*, and *resolved*. Initially, the state of every port is *unresolved*; once a port has been identified as *transparent*, its state is changed to *transparent*; after an output port has been resolved using Equation (5.3), its state is changed to *resolved*.

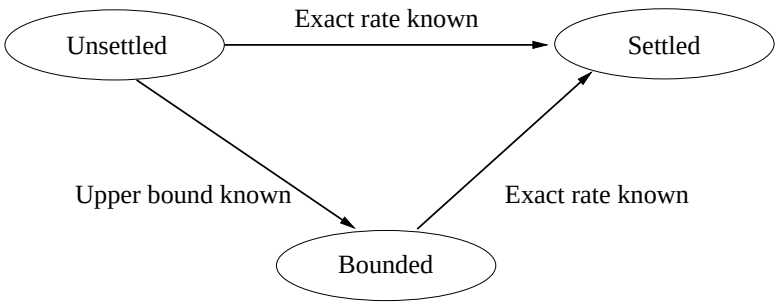


Figure 5.4: State Transition of a Flow Variable

The sequential algorithm consists of three phases: *rule-based flow update computation*, *reduced graph generation*, *fixed point iterations and residual flow update computation*. In order to explain the algorithm, we use a simple example to illustrate how it works. The network topology is shown in Figure 5.5; in the network, there are 8 output ports, q_0 - q_7, and 7 flows, f_0 - f_6. In the network, the path of flow f_0 is obviously not the shortest because the alternative one that does not visit port q_0 is one hop shorter. It is intentionally so configured to show that routing decisions in our approach are flexible. After the time-step setup, every port in the topology is in the *unresolved* state; $\lambda^{(in)}_{f_0,q_7}$, $\lambda^{(in)}_{f_1,q_7}$, $\lambda^{(in)}_{f_2,q_2}$, $\lambda^{(in)}_{f_3,q_3}$, $\lambda^{(in)}_{f_4,q_4}$, $\lambda^{(in)}_{f_5,q_5}$, and $\lambda^{(in)}_{f_6,q_6}$ are all in the *settled* state; $\lambda^{(in)}_{f_0,q_0}$, $\lambda^{(in)}_{f_0,q_1}$, $\lambda^{(in)}_{f_1,q_1}$, $\lambda^{(in)}_{f_1,q_2}$, $\lambda^{(in)}_{f_2,q_3}$, $\lambda^{(in)}_{f_3,q_4}$, $\lambda^{(in)}_{f_3,q_5}$, $\lambda^{(in)}_{f_5,q_0}$, and $\lambda^{(in)}_{f_3,q_6}$ are all in the *unsettled* state. We drop the discretized time in the above notations without causing any confusion.

Phase I: Rule-Based Flow Update Computation. In this phase, the goal is to settle as many flow variables in $\vec{\boldsymbol{\lambda}}$ as possible based on a set of prioritized rules. We use the same notations in Section 5.3 and Table 4.1.

Rule 1 is illustrated in Table 5.1. The pre-condition to trigger this rule is that there exists a port whose state is still *unresolved* but all its input flow variables have been settled. When this condition is satisfied, this port can be resolved: we change its state to *resolved*, and for each flow that traverses it, we compute its departure rate and use that to settle the corresponding input flow variable at the next output port on its path if it does not disappear into a sink.

Rule 2 illustrated in Table 5.2 still considers an output port in the *unresolved* state. It says that if all the input flow variables of this port are not in the *unsettled* state and its aggregate input rate does not exceed the link bandwidth associated with this output port, then this port can be identified as *transparent*. The pre-condition requires that every input flow variable must be in the *settled* or *bounded* state; otherwise, we cannot establish the upper bound on the aggregate input rate into this port. In addition, note that if all the input flow variables are in the *settled* state, Rule 1 can be applied on this port because it has higher priority. As a new output port is identified as *transparent*, we first change its state to *transparent*, and then for every input flow variable in the *settled* state, if it does not disappear into a sink after departing from the port, settle the corresponding input flow variable at the next output port on its path.

Precondition	$\exists q \in Q\, s.t.$ $S(q) = unresolved \wedge$ $\forall f \in F_q(t_k)\{S(\lambda_{f,q}^{(in)}(t_k) = settled)\}$
Action	$S(q) \leftarrow resolved$ $\forall f \in F_q(t_k)\, s.t.\, \Pi(f,q) \in Q$ $\{S[\lambda_{f,\Pi(f,q)}(t_k)] \leftarrow settled\,,$ $\lambda_{f,\Pi(f,q)}^{(in)}(t_k) \leftarrow \lambda_{f,q}^{(in)}(t_k) \times \min\{1, \mu_q/\Lambda_q^{(in)}(t_k)\}$

Table 5.1: Rule 1

Precondition	$\exists q \in Q\, s.t.$ $S(q) = unresolved\ \wedge$ $\sim (\exists f \in F_q(t_k)\, s.t.\, S(\lambda_{f,q}^{(in)}(t_k)) = unsettled\,)\ \wedge$ $\Lambda_q^{(in)}(t_k) \leq \mu_q\ \wedge$ $\exists f \in F_q(t_k)\, s.t.\, S(\lambda_{f,q}^{(in)}(t_k)) = bounded\,\}$
Action	$S(q) \leftarrow transparent\,,$ $\forall f \in F_q(t_k)\, s.t.\, S(\lambda_{f,q}^{(in)}(t_k)) = settled \wedge \Pi(f,q) \in Q$ $\{S[\lambda_{f,\Pi(f,q)}^{(in)}(t_k)] \leftarrow settled\,,$ $\lambda_{f,\Pi(f,q)}^{(in)}(t_k) \leftarrow \lambda_{f,q}^{(in)}(t_k)\}$

Table 5.2: Rule 2

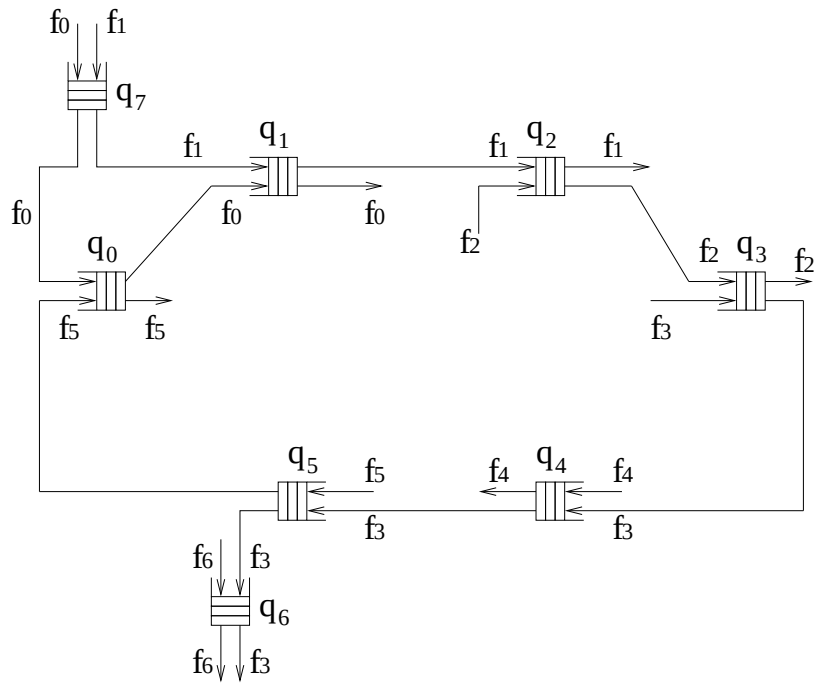


Figure 5.5: An Eight-Port Topology

Rule 3 is illustrated in Table 5.3. It is applied when a new input flow variable is settled from a *bounded* state at an output port in the *transparent* state. Because a transparent port can serve all its input traffic, the settled arrival rate into this port can be propagated downstream without any loss. If this flow does not disappear into a sink after departing from this port, the settled arrival rate can be used to settle the corresponding input flow variable at the next output port on its path.

Rule 4 still considers an output port in the *transparent* state. It is applied when an upper bound is established on an input flow variable in the *unsettled* state or a tighter upper bound is found on an input flow variable already in the *bounded* state. Because a transparent port has sufficient bandwidth to serve all its input traffic, the departure rate of this flow is equal to its arrival rate. Therefore, the upper bound on its arrival rate into the port can also be applied on its departure rate from the port. If this flow does not disappear into a sink after departing from the port, then the new upper bound can be used to bound the corresponding input flow variable at the next output port on its path.

In order to illustrate Rule 5 and Rule 6, we add a new notation. $R_q^{(in)}(t_k)$ is used to denote the sum of all the arrival rates into port q that have already been settled. Apparently, we have

$$R_q^{(in)}(t_k) \leq \Lambda_q^{(in)}(t_k). \tag{5.11}$$

Rule 5 is present in Table 5.5. It describes when an input flow variable is settled at the port, how the upper bound is set on the arrival rate into the next output port. The following lemma is

Precondition	$\exists q \in Q\, s.t.$ $S(q) = transparent \wedge$ $\exists f \in F_q(t_k)\, s.t.$ $S(\lambda^{(in)_{f,q}(t_k)}) = settled \wedge$ $\Pi(f,q) \in Q \wedge$ $\sim (S(\lambda^{(in)}_{f,\Pi(f,q)}(t_k)) = settled)$
Action	$S(\lambda^{(in)}_{f,\Pi(f,q)}(t_k)) \leftarrow settled,$ $\lambda^{(in)}_{f,\Pi(f,q)}(t_k) \leftarrow \lambda^{(in)}_{f,q}(t_k)$

Table 5.3: Rule 3

Precondition	$\exists q \in Q\, s.t.$ $S(q) = transparent \wedge$ $\exists f \in F_q(t_k)\, s.t.$ $S(\lambda^{(in)_{f,q}(t_k)}) = bounded \wedge$ $\Pi(f,q) \in Q \wedge$ $(S(\lambda^{(in)}_{f,\Pi(f,q)}(t_k)) = unsettled \vee \lambda^{(in)}_{f,\Pi(f,q)}(t_k) > \lambda^{(in)}_{f,q}(t_k))$
Action	$S(\lambda^{(in)}_{f,\Pi(f,q)}(t_k)) \leftarrow bounded,$ $\lambda^{(in)}_{f,\Pi(f,q)}(t_k) \leftarrow \lambda^{(in)}_{f,q}(t_k)$

Table 5.4: Rule 4

Precondition	$\exists q \in Q\, s.t.$ $S(q) = unresolved \wedge$ $\exists f \in F_q(t_k)\, s.t.$ $S(\lambda_{f,q}^{(in)}(t_k)) = settled \wedge$ $\Pi(f,q) \in Q \wedge$ $(S(\lambda_{f,\Pi(f,q)}^{(in)}(t_k)) = unsettled \vee$ $\lambda_{f,\Pi(f,q)}^{(in)}(t_k) > \lambda_{f,q}^{(in)}(t_k) \times \min\{1, \mu_q / R_q^{(in)}(t_k)\}\,)$
Action	$S(\lambda_{f,\Pi(f,q)}^{(in)}(t_k)) \leftarrow bounded\,,$ $\lambda_{f,\Pi(f,q)}^{(in)}(t_k) \leftarrow \lambda_{f,q}^{(in)}(t_k) \times \min\{1, \mu_q / R_q^{(in)}(t_k)\}$

Table 5.5: Rule 5

established for this purpose.

Lemma 5 *Given an input flow variable with a settled rate λ at port q, its departure rate from port q is no greater than $\lambda \times \min\{1, \mu_q / R_q^{(in)}(t_k)\}$.*

Proof: If $R_q^{(in)}(t_k) \leq \mu_q$, then the departure rate of the flow does not exceed its arrival rate λ. If $R_q^{(in)}(t_k) > \mu_q$, then $\Lambda_q^{(in)}(t_k) \geq R_q^{(in)}(t_k) > \mu_q$ from Equation (5.11); the departure rate, $\lambda \times \mu_q / \Lambda_q^{(in)}(t_k)$, must be no greater than $\lambda \times \mu_q / R_q^{(in)}(t_k)$. $\square$

Lemma 5 tells us how to compute the upper bound on the departure rate for a settled input flow variable, when we have only partial knowledge with respect to other input flow variables. If the flow does not disappear into a sink, we can use this upper bound on the departure rate to bound the corresponding input flow variable at its next output port.

Rule 6, illustrated in Table 5.5, is a complementary to Rule 5. It describes given an input flow variable in the *bounded* state, how the upper bound should be set on the arrival rate into the next output port. The following lemma is established for this purpose.

Lemma 6 *Given an input flow variable with a bounded rate λ at port q, its departure rate from port q will be no greater than $\lambda \times \min\{1, \mu_q / (\lambda + R_q^{(in)}(t_k))\}$.*

Proof: We use λ' to denote the true arrival rate of the flow into port q, and λ'' to denote the true departure rate of the flow from port q. Then, $\lambda' \leq \lambda$. We distinguish three cases.

Precondition	$\exists q \in Q\ s.t.$ $S(q) = unresolved \wedge$ $\exists f \in F_q(t_k)\ s.t.$ $S(\lambda^{(in)}_{f,q}(t_k)) = bounded \wedge$ $\Pi(f,q) \in Q\ \wedge$ $(S(\lambda^{(in)}_{f,\Pi(f,q)}(t_k)) =\ unsettled\ \vee$ $\lambda^{(in)}_{f,\Pi(f,q)}(t_k) > \lambda^{(in)}_{f,q}(t_k) \times \min\{1, \mu_q/(R^{(in)}_q(t_k) + \lambda^{(in)}_{f,q}(t_k))\}\ )$
Action	$S(\lambda^{(in)}_{f,\Pi(f,q)}(t_k)) \leftarrow\ bounded\,,$ $\lambda^{(in)}_{f,\Pi(f,q)}(t_k) \leftarrow \lambda^{(in)}_{f,q}(t_k) \times \min\{1, \mu_q/(R^{(in)}_q(t_k) + \lambda^{(in)}_{f,q}(t_k))\}$

Table 5.6: Rule 6

- **Case 1:** if $\lambda + R^{(in)}_q(t_k) \leq \mu_q$, then the departure rate λ'' satisfies the following:

$$\lambda'' \leq \lambda' \leq \lambda \tag{5.12}$$

- **Case 2:** if $\lambda + R^{(in)}_q(t_k) > \mu_q$ and $\Lambda^{(in)}_q(t_k) > \mu_q$, then the departure rate λ'' satisfies the following:

$$\begin{aligned}
\lambda'' &= \lambda' \times \frac{\mu_q}{\Lambda^{(in)}_q(t_k)} \\
&\leq \lambda' \times \frac{\mu_q}{\lambda' + R^{(in)}_q(t_k)} \quad (\text{ because } \Lambda^{(in)}_q(t_k) \geq \lambda' + R^{(in)}_q(t_k)\,) \\
&= \frac{\mu_q}{1 + \frac{R^{(in)}_q(t_k)}{\lambda'}} \\
&\leq \frac{\mu_q}{1 + \frac{R^{(in)}_q(t_k)}{\lambda}} \quad (\text{ because } \lambda' \leq \lambda\,) \\
&= \lambda \times \frac{\mu_q}{\lambda + R^{(in)}_q(t_k)}.
\end{aligned} \tag{5.13}$$

- **Case 3:** if $\lambda + R_q^{(in)}(t_k) > \mu_q$ and $\Lambda_q^{(in)}(t_k) \leq \mu_q$, then the departure rate λ'' satisfies the following:

$$\begin{aligned}
\lambda'' &= \lambda' \\
&\leq \mu_q - R_q^{(in)}(t_k) \\
&\leq \mu_q - R_q^{(in)}(t_k) \times \frac{\mu_q}{R_q^{(in)}(t_k) + \lambda} \quad (\text{ because } \lambda + R_q^{(in)}(t_k) > \mu_q\ ) \\
&= \lambda \times \frac{\mu_q}{\lambda + R_q^{(in)}(t_k)}. \qquad (5.14)
\end{aligned}$$

Combining all three cases, we prove the lemma. □

Lemma 6 informs us how to compute the upper bound on the departure rate for an input flow variable in the *bounded* state. If the flow does not disappear into a sink, we can use this upper bound on the departure rate to bound the corresponding input flow variable at its next output port.

Note that the ultimate objective of the algorithm is, actually, to resolve every output port in the topology under consideration. Rule 1 has the highest priority because every time it is applied, a new output port is resolved and some flow variables in $\vec{\boldsymbol{\lambda}}$ may be settled. Although Rule 2 does not resolve a port, it identifies a transparent port and propagates its settled flow rates downstream, which possibly leads to new ports found applicable to Rule 1. Rule 3 neither resolves a port nor identifies a transparent port, but it propagates already-settled flow rates through a transparent port. The next 3 rules establish upper bounds on flow variables. Because the first 3 rules have higher priority, flow variables tend to be settled before the last 3 rules are applied. Once no port is applicable to the first 3 rules, we try the last 3 rules, in the hope of identifying more transparent ports with tighter upper bounds.

Let r_i denote Rule i, where $1 \leq i \leq 6$. In the implementation, for each rule r_i, we maintain a list of ports, denoted by L_{r_i}. The ports on list L_{r_i} must satisfy the precondition to rule r_i. After the time step setup, we check all the ports that have input flows: if a port q is the ingress port of all the flows that traverse it, it is added onto list L_{r_1}. We then iterate over the ports on the 6 lists. Each time, we grab an arbitrary port q from the list with the highest priority and process it as the corresponding rule, say rule r, indicates. After the port is processed, we remove it from list L_r. Because we update the states of the input flow variables at downstream ports, some of them may satisfy the precondition to a rule. If we find such a port, we put it onto the corresponding port list if it is not there yet. The above iteration terminates when all the 6 port lists are empty. At this time, Phase I finishes.

In order to check whether a port satisfies the precondition to a rule in constant time, we need to be careful in the implementation. Some auxiliary variables are introduced for this purpose. At each port, we keep both the number of input flow variables in the *settled* state and the number of input flow variables in the *bounded* state. After updating the state of one or more input flow variables at a port, we update these two variables correspondingly. We can then compare the number of settled input flow variables and the total number of input flow variables at this port and decide whether Rule 1 is applicable. If Rule 1 cannot be applied, we can check whether all the input flow variables are in either *settled* or *bounded* state and then determine whether Rule 2 is applicable. With aid of

the two auxiliary variables, we can check whether a port satisfies the precondition to Rule 1 or Rule 2 in constant time.

Note that for each of the last four rules, its precondition requires that at least one input flow variable satisfy certain conditions. Hence, for each of these rules, we use a list to keep all the input flow variables that satisfy these conditions. After we process a port and update the state of an input flow variable at the corresponding downstream port, we check whether that flow variable satisfies the conditions in the precondition to any of the last four rules. If so, we put the flow variable onto the corresponding list. When a port that satisfies one of these rules is processed, we choose an arbitrary flow variable from the corresponding list, remove it from there, and process it as the rule's action indicates. In this way, it is unnecessary to traverse the states of all the input flow variables and check whether there exists an input flow variable that satisfies certain conditions.

In order to apply Rule 5 or 6, we need to know the sum of all the input flow rates that are settled so far. We introduce another variable to keep this value at each port. Once an input flow variable is settled, we add the settled input rate onto this variable. Hence, we need only constant time to obtain the sum of all the input flow rates that are settled.

From the above discussion, we have the following property:

Lemma 7 *After a port is processed, if there exists at least one unprocessed port in Phase I, we need only constant time to decide which port can be processed next.*

The following lemma tells us the possible states which a flow variable may have after Phase I.

Lemma 8 *After Phase I, every flow variable must be in the settled or bounded state.*

Proof: We prove it by contradiction. Suppose that flow variable λ is in the *unsettled* state after Phase I. We search the first flow variable that is not in the *unsettled* state along the upstream direction on the flow's path. There must exist such a flow variable because at least the flow variable corresponding to the ingress rate of the flow is in the *settled* state. Let λ' be this flow variable. Suppose that λ' belongs to port q, and the next port on the flow's path is p. If port q is in the *resolved* state, then the corresponding flow variable at port p must be settled from Rule 1; if port q is in the *transparent* state, then either Rule 3 or Rule 4 can be applied on flow variable λ'; if port q is in the *unresolved* state, then either Rule 5 or Rule 6 can be applied on flow variable λ'. In either case, a contradiction follows. □

We use the example network in Figure 5.5 to illustrate the application of the 6 rules. Given the two configurations shown in Table 5.7, our algorithm works as follows in Phase I:

- **Configuration 1:** Table 5.8 presents how the rule-based flow update computation process works on this configuration. Step 1 applies Rule 1 to resolve port q_7. Step 2 applies Rule 5 to compute the upper bound on $\lambda^{(in)}_{f_0,q_1}$. Because $\lambda^{(in)}_{f_0,q_1} + \lambda^{(in)}_{f_1,q_1} \leq \mu_{q_7}$, port q_1 can be identified as transparent; hence, the settled arrival rate of flow f_1 into port q_1 can be propagated through it and then used to settle $\lambda^{(in)}_{f_1,q_2}$. Thereafter, Rule 1 can be used to resolve ports q_2, q_3, q_4, q_5, q_6, and q_0 successively.

Configuration 1	Configuration 2
$\lambda^{(in)}_{f_0,q_7} + \lambda^{(in)}_{f_1,q_7} \leq \mu_{q_7}$	$\lambda^{(in)}_{f_0,q_7} + \lambda^{(in)}_{f_1,q_7} \leq \mu_{q_7}$
$\lambda^{(in)}_{f_0,q_5} + \lambda^{(in)}_{f_5,q_5} > \mu_{q_0}$	$\lambda^{(in)}_{f_0,q_5} + \lambda^{(in)}_{f_5,q_5} > \mu_{q_0}$
$\lambda^{(in)}_{f_0,q_7} + \lambda^{(in)}_{f_1,q_7} \leq \mu_{q_1}$	$\lambda^{(in)}_{f_0,q_7} + \lambda^{(in)}_{f_1,q_7} > \mu_{q_1}$
$\lambda^{(in)}_{f_2,q_2} + \lambda^{(in)}_{f_1,q_7} > \mu_{q_2}$	$\lambda^{(in)}_{f_2,q_2} + \lambda^{(in)}_{f_1,q_7} > \mu_{q_2}$
$\lambda^{(in)}_{f_2,q_2} + \lambda^{(in)}_{f_3,q_3} > \mu_{q_3}$	$\lambda^{(in)}_{f_2,q_2} + \lambda^{(in)}_{f_3,q_3} > \mu_{q_3}$
$\lambda^{(in)}_{f_3,q_3} + \lambda^{(in)}_{f_4,q_4} > \mu_{q_4}$	$\lambda^{(in)}_{f_3,q_3} + \lambda^{(in)}_{f_4,q_4} \leq \mu_{q_4}$
$\lambda^{(in)}_{f_5,q_5} + \lambda^{(in)}_{f_3,q_3} > \mu_{q_5}$	$\lambda^{(in)}_{f_5,q_5} + \lambda^{(in)}_{f_3,q_3} > \mu_{q_5}$
$\lambda^{(in)}_{f_3,q_3} + \lambda^{(in)}_{f_6,q_6} > \mu_{q_6}$	$\lambda^{(in)}_{f_3,q_3} + \lambda^{(in)}_{f_6,q_6} > \mu_{q_6}$

Table 5.7: Two Configurations

- **Configuration 2:** Table 5.9 presents how the rule-based flow update computation process works on this configuration. Step 1 applies Rule 1 to resolve port q_7. Step 2 applies Rule 5 to compute the upper bound on $\lambda^{(in)}_{f_0,q_1}$. Because $\lambda^{(in)}_{f_0,q_1} + \lambda^{(in)}_{f_1,q_1} > \mu_{q_7}$, port q_1 cannot be identified as transparent as on the previous configuration. Step 3 applies Rule 5 to compute the upper bound on $\lambda^{(in)}_{f_1,q_2}$; Step 4 applies Rule 5 to compute the upper bound on $\lambda^{(in)}_{f_2,q_3}$; Step 5 applies Rule 5 to compute the upper bound on $\lambda^{(in)}_{f_3,q_4}$. In step 6, because $\lambda^{(in)}_{f_4,q_4} + \lambda^{(in)}_{f_3,q_4} \leq \mu_{q_4}$, port q_4 is identified as transparent. Step 7 uses Rule 4 to compute the upper bound on $\lambda^{(in)}_{f_3,q_5}$. The final step applies Rule 5 to compute the upper bound on $\lambda^{(in)}_{f_3,q_6}$ and $\lambda^{(in)}_{f_5,q_0}$.

Both Tables 5.8 and 5.9 provide only a sample execution path for each configuration because after some steps there exist multiple choices on which ports should be processed. With configuration 1, all ports in the topology are resolved after Phase I; it is evident that with aid of transparency of port q_1, the circular dependence among flow variables in the network can be entirely removed. When this occurs, the algorithm stops after Phase I. With configuration 2, however, all ports but port q_7 are still unresolved; although port q_4 is also identified as transparent, it does not help reduce the whole circular dependence. When there still exist some unresolved ports, the algorithm proceeds into Phase II.

Phase II: Dependence Graph Generation. Owing to circular dependence, some output ports still remain unresolved after Phase I. As discussed in Section 5.4.3, the fixed point algorithm stands

Step	Rule	Processing
1	1	$S(q_7) \leftarrow resolved, S(\lambda^{(in)}_{f_0,q_0}) \leftarrow settled, S(\lambda^{(in)}_{f_1,q_1}) \leftarrow settled$
2	5	$S(\lambda^{(in)}_{f_0,q_1}) \leftarrow bounded, \lambda^{(in)}_{f_0,q_1} \leftarrow \lambda^{(in)}_{f_0,q_0}$
3	2	$S(q_1) \leftarrow transparent, S(\lambda^{(in)}_{f_1,q_2}) \leftarrow settled, \lambda^{(in)}_{f_1,q_2} \leftarrow \lambda^{(in)}_{f_1,q_1}$
4	1	$S(q_2) \leftarrow resolved, S(\lambda^{(in)}_{f_2,q_3}) \leftarrow settled,$
		$\lambda^{(in)}_{f_2,q_3} \leftarrow \lambda^{(in)}_{f_2,q_2} \times \min\{1, \mu_{q_2}/\Lambda^{(in)}_{q_2}\}$
5	1	$S(q_3) \leftarrow resolved, S(\lambda^{(in)}_{f_3,q_4}) \leftarrow settled,$
		$\lambda^{(in)}_{f_3,q_4} \leftarrow \lambda^{(in)}_{f_3,q_3} \times \min\{1, \mu_{q_3}/\Lambda^{(in)}_{q_3}\}$
6	1	$S(q_4) \leftarrow resolved, S(\lambda^{(in)}_{f_3,q_5}) \leftarrow settled,$
		$\lambda^{(in)}_{f_3,q_5} \leftarrow \lambda^{(in)}_{f_3,q_4} \times \min\{1, \mu_{q_4}/\Lambda^{(in)}_{q_4}\}$
7	1	$S(q_5) \leftarrow resolved, S(\lambda^{(in)}_{f_3,q_6}) \leftarrow settled,$
		$\lambda^{(in)}_{f_3,q_6} \leftarrow \lambda^{(in)}_{f_3,q_5} \times \min\{1, \mu_{q_5}/\Lambda^{(in)}_{q_5}\}$
		$S(\lambda^{(in)}_{f_5,q_0}) \leftarrow settled, \lambda^{(in)}_{f_5,q_0} \leftarrow \lambda^{(in)}_{f_5,q_5} \times \min\{1, \mu_{q_5}/\Lambda^{(in)}_{q_5}\}$
8	1	$S(q_0) \leftarrow resolved, S(\lambda^{(in)}_{f_0,q_1}) \leftarrow settled,$
		$\lambda^{(in)}_{f_0,q_1} \leftarrow \lambda^{(in)}_{f_0,q_0} \times \min\{1, \mu_{q_0}/\Lambda^{(in)}_{q_0}\}$

Table 5.8: Flow Update Computation for the Network in Figure 5.5 (Configuration 1)

Step	Rule	Processing
1	1	$S(q_7) \leftarrow resolved, S(\lambda^{(in)}_{f_0,q_0}) \leftarrow settled, S(\lambda^{(in)}_{f_1,q_1}) \leftarrow settled$
2	5	$S(\lambda^{(in)}_{f_0,q_1}) \leftarrow bounded, \lambda^{(in)}_{f_0,q_1} \leftarrow \lambda^{(in)}_{f_0,q_0}$
3	5	$S(\lambda^{(in)}_{f_1,q_2}) \leftarrow bounded, \lambda^{(in)}_{f_1,q_2} \leftarrow \lambda^{(in)}_{f_1,q_1}$
4	5	$S(\lambda^{(in)}_{f_2,q_3}) \leftarrow bounded, \lambda^{(in)}_{f_2,q_3} \leftarrow \lambda^{(in)}_{f_2,q_2}$
5	5	$S(\lambda^{(in)}_{f_3,q_4}) \leftarrow bounded, \lambda^{(in)}_{f_3,q_4} \leftarrow \lambda^{(in)}_{f_3,q_3}$
6	2	$S(q_4) \leftarrow transparent$
7	4	$S(\lambda^{(in)}_{f_3,q_5}) \leftarrow bounded, \lambda^{(in)}_{f_3,q_5} \leftarrow \lambda^{(in)}_{f_3,q_4}$
8	5	$S(\lambda^{(in)}_{f_3,q_6}) \leftarrow bounded, \lambda^{(in)}_{f_3,q_6} \leftarrow \lambda^{(in)}_{f_3,q_5}$
		$S(\lambda^{(in)}_{f_5,q_0}) \leftarrow bounded, \lambda^{(in)}_{f_5,q_0} \leftarrow \lambda^{(in)}_{f_5,q_5}$

Table 5.9: Flow Update Computation for the Network in Figure 5.5 (Configuration 2)

out as the only practical solution to determining the rates of those unsettled flow variables in $\vec{\boldsymbol{\lambda}}$. In order to reduce the computation cost on fixed point iterations, we try to minimize the number of flow variables that are involved. The example network with the second configuration is used to build some intuition on which flow variables should be considered. After Phase I, we still do not know the exact rates of flow variables $\lambda^{(in)}_{f_0,q_1}$, $\lambda^{(in)}_{f_1,q_2}$, $\lambda^{(in)}_{f_2,q_3}$, $\lambda^{(in)}_{f_3,q_4}$, $\lambda^{(in)}_{f_3,q_5}$, $\lambda^{(in)}_{f_3,q_6}$, and $\lambda^{(in)}_{f_5,q_0}$. Remember that port q_4 has been identified as transparent, which suggests that $\lambda^{(in)}_{f_3,q_4}$ must be equal to $\lambda^{(in)}_{f_3,q_5}$. On the other hand, $\lambda^{(in)}_{f_3,q_6}$ actually does not stay in any cycle: settling it entirely depends on the resolution of port q_5. These two observations lead to the following optimizations:

- **Optimization 1:** We exclude every flow variable associated with a transparent port from the fixed point iterations;
- **Optimization 2:** We exclude every flow variable that is not on any circular dependency from the fixed point iterations;

Now we discuss how to implement these two optimizations by constructing dependence graph G. First we construct set V, which contains all the output ports in the *unresolved* state:

$$V = \{q \mid q \in Q \wedge S(q) = unresolved\}. \tag{5.15}$$

Note that some output ports whose states are *transparent* may still have some unsettled input flow variables, but they are excluded from set V based on the first optimization. The ports in V form the vertices in dependence graph G. We use E to denote the set of edges in graph G. This set is initialized to be empty. Every edge e is associated with a set of flow variables, denoted by $F_G(e)$. For every port q in V, we process each of its input flow variables as follows: search the first output port q' downstream on the flow's path that is also included in set V; if such a port can be found, a directed edge from port q to port q', denoted by $e(q, q')$, is added to E (if it is not in E yet) and the corresponding input flow variable at the receiving port is added into set $F_G(e(q, q'))$. After all ports in V have been processed, the resulting graph G has the following property: no port in graph G has indegree 0. If there exists a port with indegree 0, then its resolution does not depend on any other port's state, and therefore, it must have already been resolved in Phase I, which contradicts it being in set V.

However, it is likely that some ports in V have outdegree 0. This happens when input flow variables associated with these ports are not in any circular dependence. In the example network, port q_6 has outdegree 0 because flow variable $\lambda^{(in)}_{f_3,q_6}$ is not in the circular dependence formed by other variables. According to the second optimization, we remove the ports with outdegree 0 from graph G and all the edges that point to them. After removing a port with outdegree 0, we decrease the outdegrees of those ports that have an edge pointing to it by 1; if the outdegree of a port becomes 0 thereby, it should also be removed from graph G. Then the above process repeats, until no port with outdegree 0 exists in dependence graph G. Phase II finishes when the process terminates. Note that when we prune the ports with outdegree 0 from the graph, no port with indegree 0 should be introduced. Hence, the following lemma should hold after Phase II:

Lemma 9 *In the final graph G, no port has indegree 0 or outdegree 0.*

As for the example network with the second configuration, the final dependence graph G is shown in Figure 5.6.

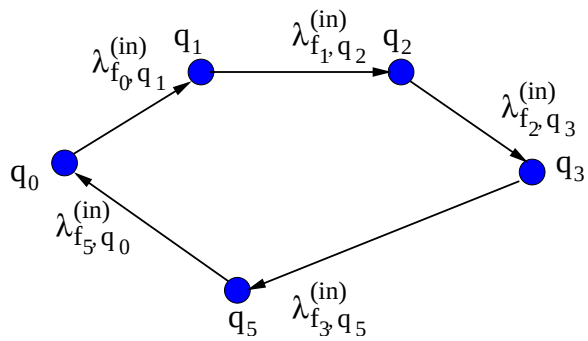


Figure 5.6: Dependence Graph for the Example Network with Configuration 2

Phase III: Fixed Point Iterations and Residual Flow Update Computation. The fixed point algorithm described before operates on all the input flow variables in the network except those that correspond to the ingress rates of flows. Our approach aims to minimize the number of flow variables involved in the fixed point iterations. We construct another vector $\vec{\boldsymbol{\lambda}}_g$ to contain all the flow variables that need to be settled by the fixed point algorithm. For every edge e in dependence graph G (i.e., $e \in E$), we put all the flow variables in $F_G(e)$ into $\vec{\boldsymbol{\lambda}}_g$. Using Equation (5.3), we can form a set of equations of variables in $\vec{\boldsymbol{\lambda}}_g$. The fixed point algorithm is then applied to find the root of this group of equations. As discussed in Section (5.4.3), the algorithm starts from an initial estimate on $\vec{\boldsymbol{\lambda}}_g$. The simplest estimate is $\vec{\mathbf{0}}$, in which all the variables in $\vec{\boldsymbol{\lambda}}_g$ are 0. This estimate may be very far from the real fixed point; a lot of iterations are, therefore, necessary before that fixed point is approached. Hence, our solution adopts an alternative estimate: since every flow variable in $\vec{\boldsymbol{\lambda}}_g$ must have an upper bound when Phase I finishes, these upper bounds on the variables in $\vec{\boldsymbol{\lambda}}_g$ form the initial estimate. Therefore, tighter upper bounds on flow variables found in Phase I may help reduce the number of iterations required in the fixed point algorithm.

The fixed point iterations terminate when the maximum relative difference between successive estimates on any flow variable in $\vec{\boldsymbol{\lambda}}_g$ is less than a tolerance ϵ. We use $\vec{\boldsymbol{\lambda}}_g^{(i)}[j]$, $i = 0, 1, 2, ...$ and $1 \leq j \leq |\vec{\boldsymbol{\lambda}}_g|$ to denote the estimate on the j-th variable in $\vec{\boldsymbol{\lambda}}_g$ after iteration i. Then, the termination criterion is

$$\max_{1 \leq j \leq |\vec{\boldsymbol{\lambda}}_g|} \frac{|\vec{\boldsymbol{\lambda}}_g^{(i+1)}[j] - \vec{\boldsymbol{\lambda}}_g^{(i+1)}[j]|}{\vec{\boldsymbol{\lambda}}_g^{(i)}[j]} \leq \epsilon. \tag{5.16}$$

Once the fixed point algorithm terminates, we change the state of each flow variable in $\vec{\boldsymbol{\lambda}}_g$ to *settled*.

In the example, $\vec{\boldsymbol{\lambda}}_g$ is constructed as

$$< \lambda^{(in)}_{f_0,q_1}, \lambda^{(in)}_{f_1,q_2}, \lambda^{(in)}_{f_2,q_3}, \lambda^{(in)}_{f_3,q_5}, \lambda^{(in)}_{f_5,q_0} > . \tag{5.17}$$

Using Equation (5.3), we form a group of equations as follows:

$$\begin{cases} \lambda^{(in)}_{f_0,q_1} &= \lambda^{(in)}_{f_0,q_0} \times \min\{1, \mu_{q_0}/(\lambda^{(in)}_{f_0,q_0} + \lambda^{(in)}_{f_5,q_0})\} \\ \lambda^{(in)}_{f_1,q_2} &= \lambda^{(in)}_{f_1,q_1} \times \min\{1, \mu_{q_1}/(\lambda^{(in)}_{f_1,q_1} + \lambda^{(in)}_{f_0,q_1})\} \\ \lambda^{(in)}_{f_2,q_3} &= \lambda^{(in)}_{f_2,q_2} \times \min\{1, \mu_{q_2}/(\lambda^{(in)}_{f_2,q_2} + \lambda^{(in)}_{f_1,q_1})\} \\ \lambda^{(in)}_{f_3,q_5} &= \lambda^{(in)}_{f_3,q_3} \times \min\{1, \mu_{q_3}/(\lambda^{(in)}_{f_3,q_3} + \lambda^{(in)}_{f_2,q_3})\} \\ \lambda^{(in)}_{f_5,q_0} &= \lambda^{(in)}_{f_5,q_5} \times \min\{1, \mu_{q_5}/(\lambda^{(in)}_{f_5,q_5} + \lambda^{(in)}_{f_3,q_3})\} \end{cases} \tag{5.18}$$

The fixed point algorithm is then applied to solve Equation (5.18), starting from an initial estimate in which every flow variable in $\vec{\boldsymbol{\lambda}}_g$ uses its upper bound obtained in Phase I. Recall that in Phase II, transparent ports and ports with outdegree 0 are excluded from V. At these ports, therefore, some input flow variables are still unsettled after the fixed point iterations. We can resume the rule-based flow update computation and resolve all the unresolved ports. In the example, $\lambda^{(in)}_{f_3,q_6}$ and $\lambda^{(in)}_{f_3,q_4}$ are not settled after the fixed point iterations. Because both input flow variables at port q_3 are settled, Rule 1 can be used to compute $\lambda^{(in)}_{f_3,q_4}$; $\lambda^{(in)}_{f_3,q_6}$ can also be calculated by applying Rule 1 on port q_5.

After Phase III, all output ports in the network have been resolved. The algorithm then ends here. In the following sections, we will discuss the issues regarding the convergence behavior, performance, and accuracy of our algorithm.

5.5.2 Convergence Behavior

If the flows in the network form circular dependencies that cannot be resolved in Phase I, the fixed point algorithm is necessary to settle the unsettled flow variables associated with these circular dependencies. As mentioned in Section 5.4.3, the convergence behavior of the fixed point iterations directly affects the performance of our algorithm. For example, if the fixed point iterations diverge, then our algorithm may never terminate and its foundation is thus severely undermined.

In mathematics, two fixed point theorems are often used to prove the existence of fixed points [30]. The *Brouwer fixed point theorem* states that every continuous function from a nonempty, compact and convex subset of $\mathbf{R}^n$ to itself must have at least one fixed point. Although elegant, the Brouwer fixed point theorem is non-constructive: it does not describe how to find the fixed points. Rather, the *Banach fixed point theorem* provides an approach to discovering fixed points for functions that are contraction mappings. A function $T : X \rightarrow X$ is said to be a *contraction mapping* if there exists a constant q with $0 \leq q < 1$ such that

$$d(T(x), T(y)) \leq q \cdot d(x, y) \tag{5.19}$$

where the real valued function $d : X \times X \rightarrow \mathbf{R}$ is a distance function or *metric*. The Banach fixed point theorem states that every contraction mapping has a unique fixed point. For an arbitrary $x^{(0)} \in X$, iterate as $x^{(n+1)} = T(x^{(n)})$. The sequence converges to the unique fixed point. Hence, the Banach fixed point theorem provides a constructive approach to finding the fixed point. In order to use the Banach fixed point theorem to guarantee convergence of our algorithm, we need to prove that function H in Equation (5.10) is a contraction mapping. As discussed in Section 5.4.3, however, $H(\cdot)$ is a multi-dimension nonlinear system of equations containing min functions, whose mathematical analysis is well known to be difficult, especially when there are a large number of variables in the system.

In [62], a related problem has been investigated in the context of multi-class networks with FIFO service disciplines. The main theorem in it establishes the existence of a unique fixed point solution for a reentrant line that traverses multiple stations whose service rates are independent of the line's stage. A reentrant line and a station resemble a flow and a port in our problem respectively. In the final remark of the paper, it is also claimed that the theorem can be extended for the cases in which multiple reentrant lines exist.

The theoretical results in [62] may be helpful in establishing the existence of a unique solution to our problem. Even so, however, we are still unclear of whether the fixed point iterations finally converge to that fixed point. Furthermore, even if they converge, how fast they converge still remain as a mystery to us; a slowly convergent algorithm may lead to poor performance because a lot of computation work is spent on the iterations before a fixed point is reached. Provided these challenges on understanding the convergence behavior of our algorithm, we put an upper bound on how many iterations are allowed in a time step. In a real network, traffic traversing across a link suffers some delay. We assume that all the edges in the dependence graph built in Phase II are associated with a universal delay d. Then, iterations in the fixed point algorithm can be considered as periodically updating the states of flow variables every d simulation time units. Recall that the time step size is Δ simulation time units. Therefore, there are at most $\lfloor \Delta/d \rfloor$ iterations permissible within one time step, where $\lfloor x \rfloor$ is the largest number no greater than x. The selection on d in our approach is given as follows. Let $d_f(e)$ denote the aggregate delay from the node where edge e starts to the node where edge e ends on flow's f's path in the real network. For each edge e in E, we associate with it the largest aggregate delay that any flow across it suffers. Then among all the edges in the dependence graph, we pick the one that has the largest aggregate delay. We choose d as follows:

$$d = \max_{e \in E} \max_{f \in F_e(G)} d_f(e). \tag{5.20}$$

The above upper bound is aimed at avoiding excessive number of iterations that the fixed point algorithm has to take before the fixed point is found or when the fixed point cannot be found through iterations. In all the following experiments, we actually find that the fixed point algorithm converges within a small number of iterations, and the upper bounds have never been reached.

In Table 5.10, we summarize four real topologies that are used in the simulation experiments in this section and the following sections. Top-1, which is illustrated in Figure 4.7, is a POP-level ATT USA backbone network. A flow is created between any pair of POPs, which thus leads to 702 flows in total. The bandwidth of each link is set to be 100Mbps. This topology later will

Topology	# nodes	# directional links	# flows	Link bandwidth
Top-1	27	88	702	100 Mbps
Top-2	244	1080	12200	2488 Mbps
Top-3	610	3012	61000	2488 Mbps
Top-4	1126	6238	168900	2488 Mbps

Table 5.10: 4 Topologies Used in the Simulations

be used in Section 5.5.3 to do performance comparison between the time-stepped coarse-grained traffic simulator and the packet-oriented simulator; a comparatively low link bandwidth enables us to finish a certain number of simulation runs with the packet-oriented simulator in a reasonable period of time. The other three topologies are obtained from the Rocketfuel project[1]. Top-2 is the router-level Exodus backbone; Top-3 consists of two router-level ISP backbones, the Exodus backbone and the Above.Net backbone, which are connected through some peering links; Top-4 further augments Top-3 by adding another router-level ISP backbone, Sprint backbone, to it. In Top-2, Top-3, and Top-4, every router picks 50 routers from its own backbone and directs a flow to each of them; it also picks 50 routers from each other backbone in the topology and directs a flow to every one of them. Hence we get the total numbers of flows shown in the table. We use the PPBP traffic model to generate the ingress rate for each flow. The Hurst parameter is 0.8. The coefficient of variance (COV), which is the ratio of the standard deviation to the mean, is a common measure of traffic burstiness [74]. In our experiments, we test two COVs, 5 and 0.5. The tolerance ϵ, which is used to determine whether fixed point iterations should be terminated (See Equation (5.16)), is 0.001.

In the experiments studying the convergence of our algorithm, we simulate Top-2, Top-3, and Top-4 for 200 time steps, each spanning 5 seconds. Figures 5.7 - 5.11 give the histogram of the number of ports on the dependence graph in Phase III of a time step, and the number of iterations used to reach fixed point solutions, on single runs of 200 time steps with 2 COVs. The simulation results can be analyzed from the following aspects:

- **The same COV, the same link utilization, different topology sizes.** Under this setting, we consider how the topology size affects the convergence behavior. When the simulated topology contains more nodes and thus more flows, more ports may appear on the dependence graph in Phase III. Although the traffic load at each port is maintained at the same level, such growth is observed to be sup-linear with the topology size, measured by the number of nodes. For example, consider the results when the link utilization is 50% and the COV is 5. The

[1] http://www.cs.washington.edu/research/networking/rocketfuel/

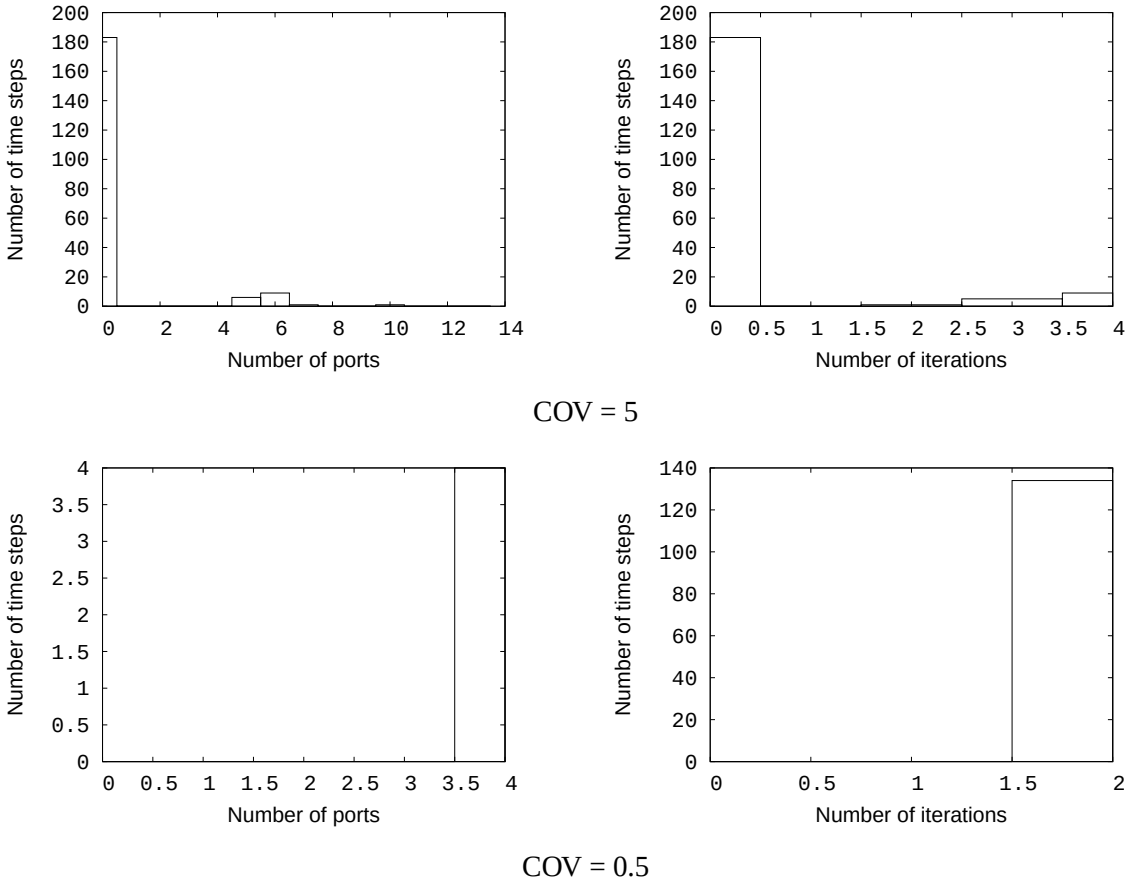


Figure 5.7: Histogram of Ports on Circular Dependencies, and Iterations per Time-step, for Top-2 20% Link Utilization

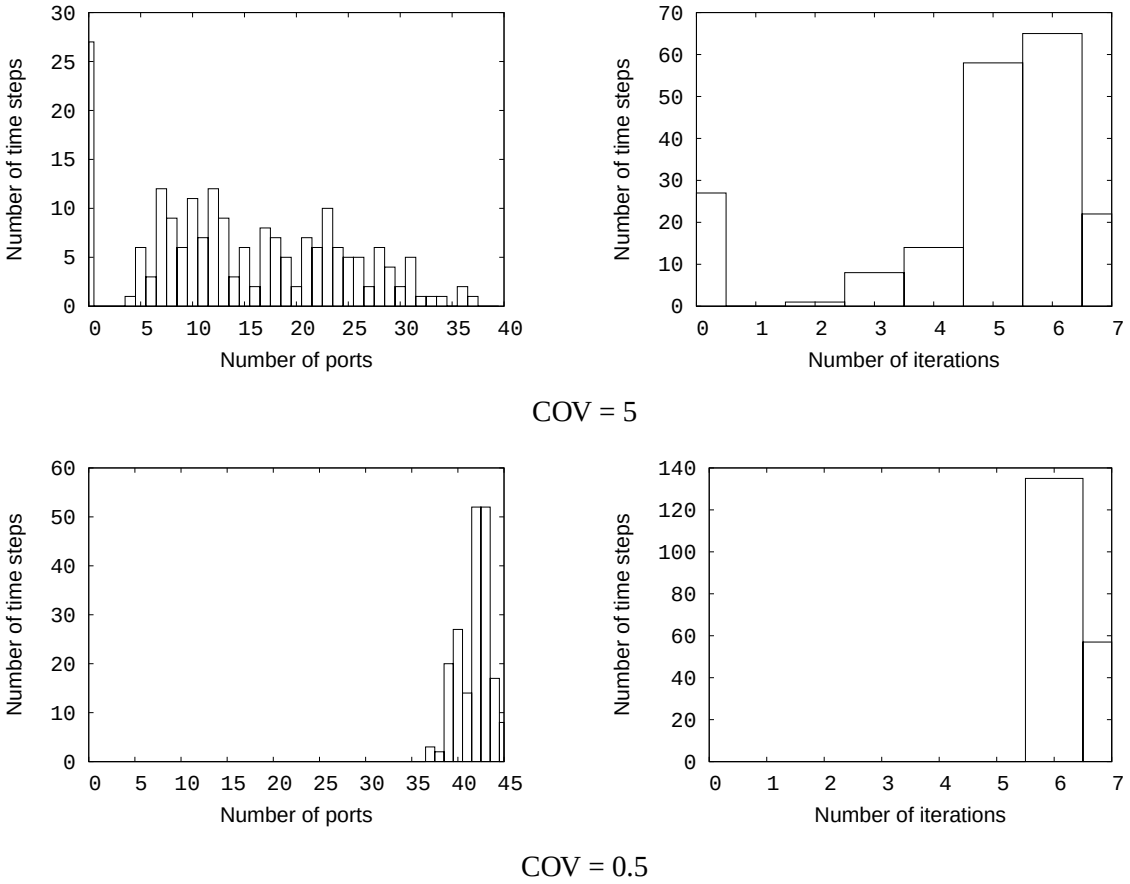


Figure 5.8: Histogram of Ports on Circular Dependencies, and Iterations per Time-step, for Top-2 50% Link Utilization

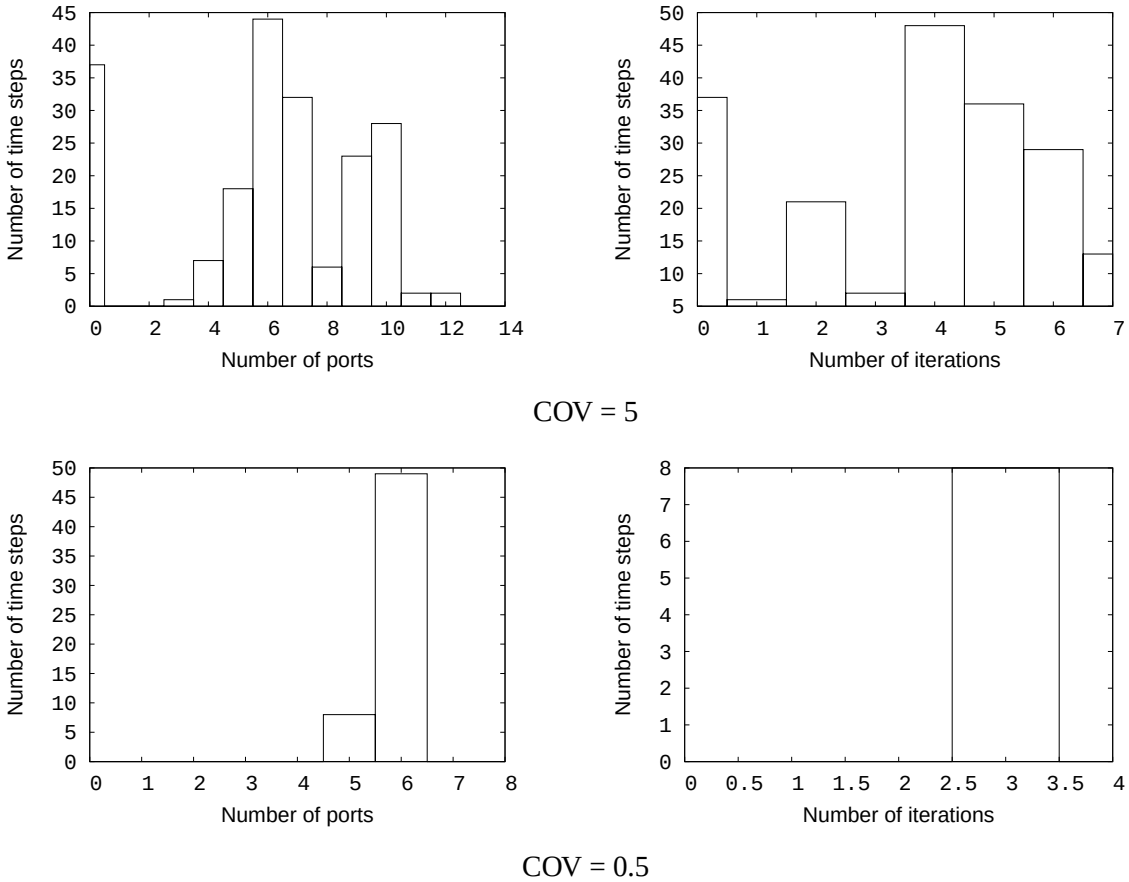


Figure 5.9: Histogram of Ports on Circular Dependencies, and Iterations per Time-Step, for Top-3 20% Link Utilization

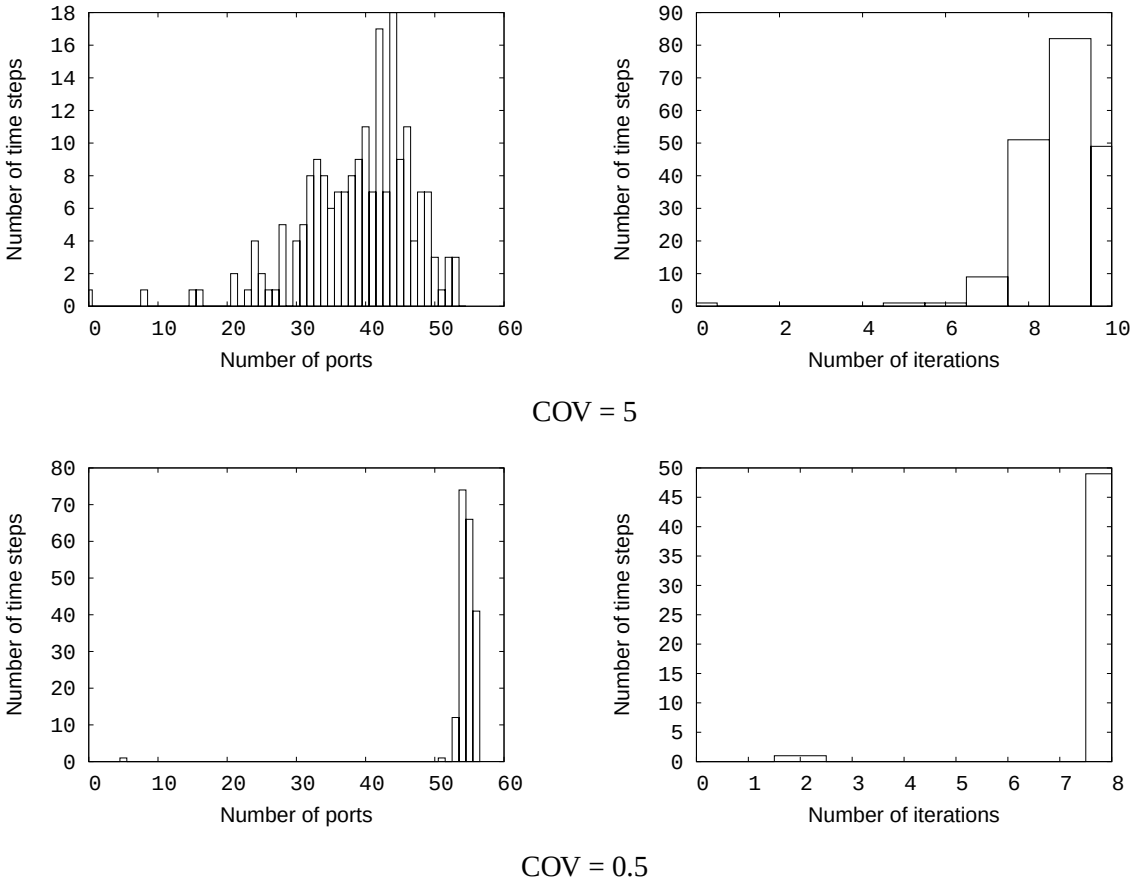
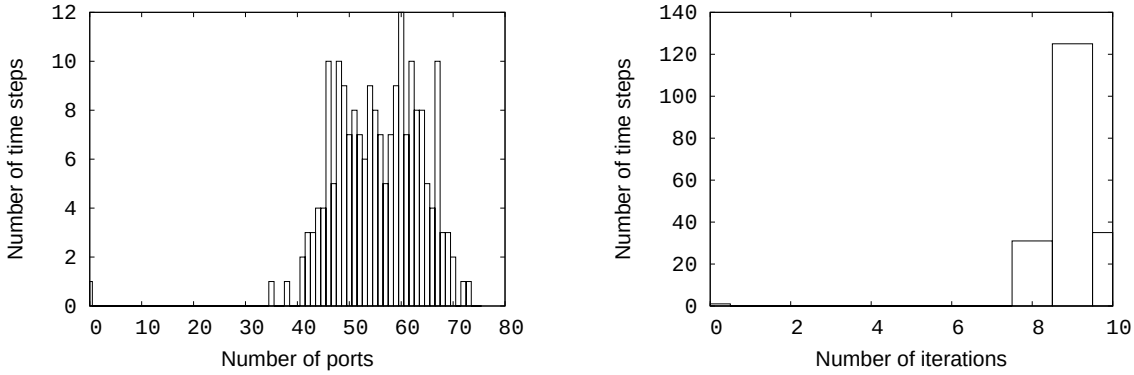
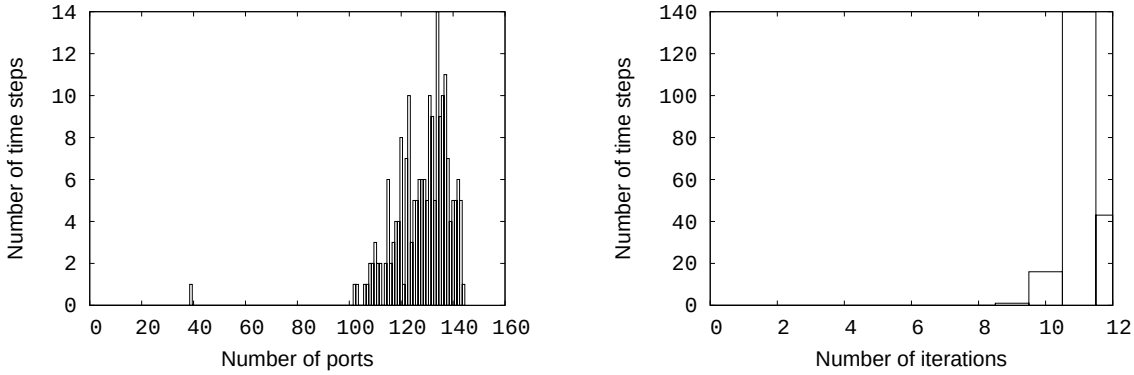


Figure 5.10: Histogram of Ports on Circular Dependencies, and Iterations per Time-Step, for Top-3 50% Link Utilization



Top-4 Backbone 20% Link Utilization



Top-4 Backbone 50% Link Utilization

Figure 5.11: Histogram of Ports on Circular Dependencies, and Iterations per Time-Step, for Top-4 (COV = 5)

average number of ports on the dependence graph in Phase III is 14 for Top-2, 39 for Top-3, and 128 for Top-4, but the sizes of Top-3 and Top-4 are 2.5 and 4.6 times as that of Top-2 respectively. This can be explained as follows. As the topology size increases, although the average link utilization is maintained at the same level, congested ports are more likely to be dependent on each other because on average more flows traverse a port. Another observation from the simulation results is that as the topology size increases, the average number of iterations required to reach a fixed point increases slightly; this is reasonable because more ports are on the dependence graph in Phase III.

- **The same COV, the same topology, different link utilizations.** The impact that the traffic load in the network has on the convergence behavior of our algorithm is clear: if the average link utilization is higher , then it is more likely for a port to be congested and thus more ports appear on the dependence graph in Phase III. This is also observed from the simulation results. For example, consider the results when the COV is 5. If the link utilization is 20%, the average number of ports on the dependence graph in Phase III corresponding to Top-2, Top-3, and Top-4 is 0, 5, and 55 respectively, but if the link utilization is 50%, the average number of ports on the dependence graph in Phase III corresponding to Top-2, Top-3, and Top-4 is 14, 39, and 128 respectively.

- **The same topology, the same link utilization, different COVs.** The COV is a metric of traffic burstiness. Its impacts on the convergence behavior of our algorithm vary with the traffic load in the topology. When the link utilization is 20%, if the COV is higher, more ports appear on the dependence graph in Phase III; when the link utilization is 50%, if the COV is higher, it seems that fewer ports appear on the dependence graph in Phase III. This observation can be explained as follows. When the link utilization is low, the traffic needs to be highly bursty to cause congestion in the network. When the link utilization is high, a certain level of burstiness is enough to cause congestion. However, if the mean is the same and the COV is relatively large, then the bursty traffic pattern generated at the ingress node of a flow has high peak rates, followed by long "silent" periods (i.e., the flow has rate 0). The peak rates, if exceeding the link bandwidth associated with the ingress port, are reduced, causing the ports on the flow's path to see less traffic. Therefore, when the link utilization and the COV are both high, it is observed that fewer ports appear on dependence graph in Phase III from the simulation results. Another observation is that when the COV is higher and the traffic is thus more bursty, the number of ports that appear on the dependence graph in Phase III over the 200 time steps vary in a larger range. This is because if the traffic is less bursty, the change on the aggregate arrival rate into a port is less dynamic across the time steps and congestion occurs more regularly in the network.

In summary, our algorithm is able to significantly reduce the number of ports in the fixed point iterations. Under any traffic load and any COV, the largest proportion of ports that appear on the dependence graph in Phase III is 4.4%, 2.8%, and 3.4% for TOP-2, TOP-3, and TOP-4 respectively. Furthermore, the fixed point solution in our algorithm can converge within a small number of iterations. In all the experiments, we have observed that at most 12 iterations are necessary to reach a solution that satisfies the termination criterion (5.16).

5.5.3 Performance

In Section 5.4, we have discussed the performance of some specific solutions in diverse settings. Corollaries 3 and 4 tell us that the fixed point algorithm does not provide the optimal performance if it is applied on some specific networks like feed-forward networks or ones that have sufficient bandwidth. Then how does our algorithm perform on these networks? Following the same notations there, we use N and M to denote the total number of flows in the network and the average number of output ports on a flow's path. The following theorem establishes the time complexity of our algorithm when applied on a feed-forward network.

Theorem 5 *Given a feed-forward network, the time complexity of the algorithm described in Section 5.5.1 is* $\Theta(N \cdot M)$.

Proof: There is no circular dependence among flow variables in a feed-forward network. Hence, all ports are resolved in Phase I. In our implementation, Rule 1 is iteratively used to resolve all the output ports in a feed-forward network because it has the highest priority. The state of every output port in the topology is read or updated for a constant number of times when it is resolved with Rule 1; correspondingly, every input flow variable is also read or updated for a constant number of times in the course of the computation. Lemma 7 tells us that after a port is processed, only constant time is needed to find the next unprocessed port. Therefore, the time complexity of the algorithm is $\Theta(N \cdot M)$. □

The next theorem establishes the time complexity of our algorithm when it is applied on a network with sufficient bandwidth.

Theorem 6 *Given a network with sufficient bandwidth, the time complexity of the algorithm described in Section 5.5.1 is* $\Theta(N \cdot M)$.

Proof: When our algorithm is used to resolve the ports in a network with sufficient bandwidth, all the flow variables are settled in Phase I because transparency can make the ingress rate of every flow be propagated to its destination. Only the first three rules are applied because they have the higher priorities than the other three. The state of each input flow variable changes exactly once from *unsettled* to *settled*. Since there are $\Theta(N \cdot M)$ input flow variables, the total computation cost on updating the states of the input flow variables is $\Theta(N \cdot M)$. On the other hand, whenever a rule is applied on a port, we propagate the settled rate of at least one input flow through it and settle the corresponding input flow variable at the next hop. The total computation cost on reading or updating the states of the ports is thus $O(N \cdot M)$. Therefore, the time complexity of the algorithm is thus $\Theta(N \cdot M)$. □

Theorems 5 and 6 tell us that on some specific networks, our algorithm requires less asymptotic computation cost than the original fixed point algorithm. It actually has the same asymptotic time complexity as the particular solutions discussed in Sections 5.4.1 and 5.4.2. However, our algorithm, like the original fixed point algorithm, has the power to solve the problem on networks involving circular dependencies.

If circular dependencies cannot be completely removed from the topology after Phase I, analyzing the time complexity of our algorithm becomes more complicated for two reasons. First, it is difficult to establish exactly how many times the upper bound of a flow variable is updated in Phase I. Our strategy is to lower the upper bound of every flow variable as much as we can, in the hope that more transparent ports can thus be exposed in Phase I; in addition, a tighter upper bound on a flow variable will provide a better initial estimate on it later for the fixed point algorithm. This strategy, however, makes it difficult to establish how many times the upper bound of a flow variable has been changed, especially in a large topology where many ports are in circular dependence. On the other hand, the performance of our algorithm is also contingent on how many ports remain unresolved after Phase I and how many iterations have been done in the fixed point algorithm. Both factors are topology-dependent, and the latter also depends on the precision tolerance ϵ in the termination criterion (5.16).

We therefore empirically investigate the performance of our algorithm under realistic topologies. The four topologies in Table 5.10 are simulated at two traffic intensity levels: 20% average link utilization and 50% average link utilization. The time step we choose is 5 seconds, which is much larger than the typical end-to-end delays even in a large network. We use the PPBP traffic model to generate the ingress rate for each flow. The Hurst parameter is 0.8. As we said in Section 5.5.2, traffic burstiness can be measured with COV. In our experiments, we test two COVs, 5 and 0.5. All the experiments are done on a 1.5GHz PC with 2Gb of memory. The average execution times per time-step for these four topologies under different traffic loads are presented in Table 5.11. Because the variations on these execution times are all quite small, we do not show the confidence interval in the table. We do not have the results for Top-4 when the COV is 0.5 because the memory required in the simulation exceeds the physical limit.

All the simulations are completed within less than 5 seconds, the time step we choose. It suggests that these simulations are finished in real time. From the results shown in Table 5.11, it seems that if the traffic is more bursty, less simulation execution time is required. This is counter-intuitive because high variation usually cause more congestion, thus putting more ports on the dependence graph in Phase III. We explain this observation as follows. When the COV is large, traffic generated from the PPBP model is actually very bursty. As said in Section 5.5.2, if the COV is larger, the peak rates in the bursts are higher, followed by longer "silent" periods. In our implementation, when a flow does not have traffic to emit at a time step, we do not push rate 0 along its path because it does not affect the computation of flow variables. Obviously, if more flows have ingress rate 0 at a time step, there are fewer flow variables that need to be settled. In addition, from the simulation results shown in Section 5.5.2, we know that when the link utilization is high, higher burstiness actually reduces the number of ports on the dependence graph in Phase III. These two observations both explain why simulations configured with higher COVs take less time to finish.

We have also done some experiments on Top-1 to obtain the relatively execution speedup of the time-stepped coarse-grained traffic simulator against a pure packet-level network simulator. The Hurst parameter is 0.8. We test two COVs: 5 and 0.5. We also vary the ingress rates of flows to obtain different average link utilizations in the network. The relative execution speedups, as a function of average link utilization, are presented in Figure 5.12. It is clear that time-stepped coarse-grained traffic simulation has achieved execution speedups at several orders of magnitude under

Topology	COV = 5		COV = 0.5	
	secs/round (20% link util)	secs/round (50% link util)	secs/round (20% link util)	secs/round (50% link util)
Top-1	0.0026	0.0026	0.0051	0.0052
Top-2	0.051	0.051	0.1988	0.2380
Top-3	0.283	0.285	1.4895	1.8740
Top-4	0.852	0.907	-	-

Table 5.11: Average Execution Time per Time-Step

both traffic loads. We have also observed that with increasing average link utilization, the speedup over the pure packet-level simulator also improves. This is because the performance of a packet-level traffic simulator is directly affected by the amount of traffic traversing through the network, but performance of the time-stepped coarse-grained traffic simulator is relatively insensitive to the absolute traffic load in the network.

From Figure 5.12, we have noticed that the relative speedup curve grows *sublinearly* as the average link utilization exceeds 50%. There are two reasons for this. First, when the traffic load increases up to a high level, the pure packet-level traffic simulator drops a portion of traffic owing to congestion. Second, heavy traffic load causes more output ports to become circularly dependent and the fixed point algorithm thus has to operate on more unsettled flow variables in Phase III.

In addition, comparing the two curves with different COVs, we can find that a larger COV leads to less relative speedup from the time-stepped coarse-grained traffic simulator. When a pure packet-level simulator is used, if the traffic is more bursty, more packets are dropped in the network, which significantly reduces the execution time. Although from Table 5.11 we know that with a larger COV, the time-stepped coarse-grained simulation also needs less execution time, it has a smaller impact on the speedup than the reduced execution time due to packet losses by the pure packet-level simulator.

5.5.4 Accuracy

In this section, we discuss the accuracy of our approach as a background traffic generator against the pure packet-oriented approach. We modify topology Top-1 by attaching an end host to each POP node in it. The background traffic is simulated with two different techniques: our time-stepped coarse-grained simulation and conventional packet-level simulation. We vary the average back-

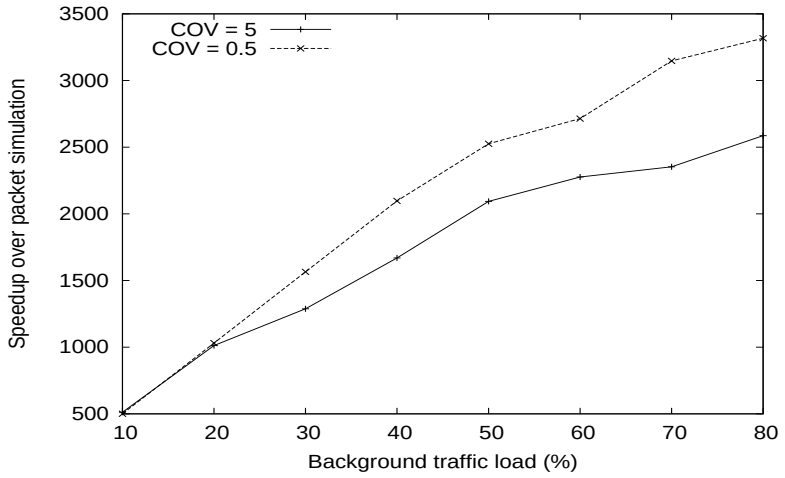


Figure 5.12: Speedup of Time-Stepped Coarse-Grained Traffic Simulator over Packet-Oriented Simulator (Top-1)

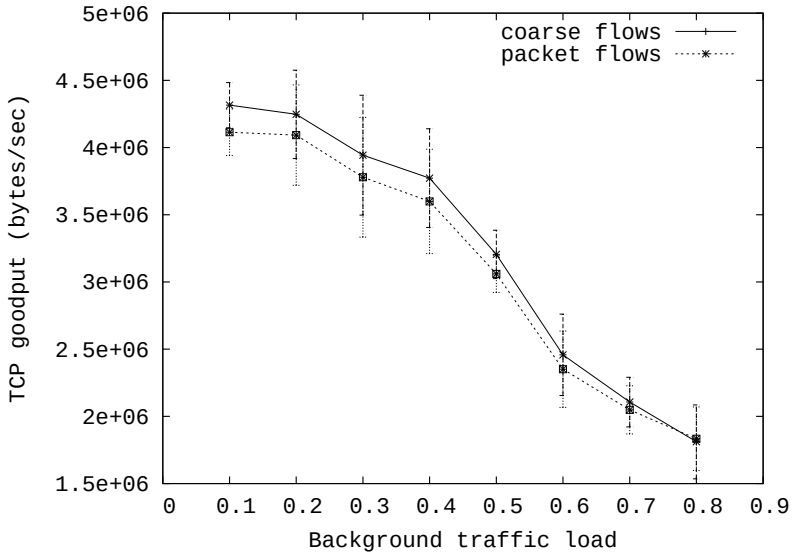


Figure 5.13: TCP Goodput (95% Confidence Interval)

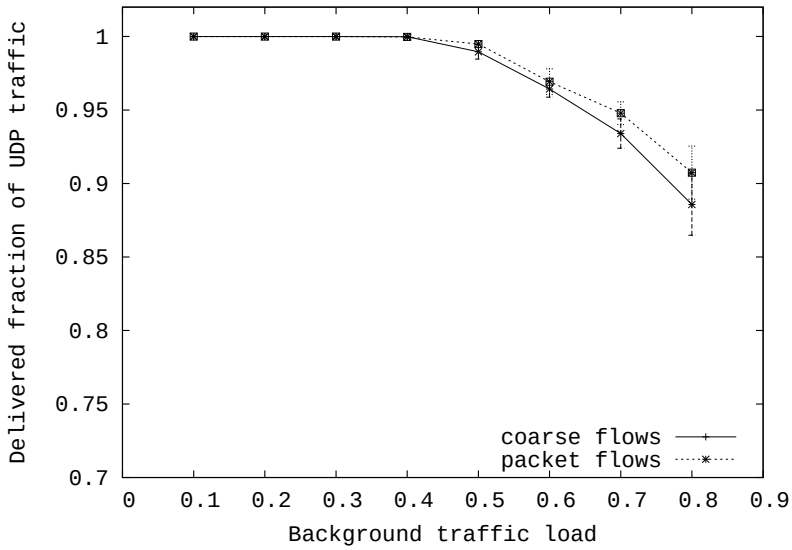


Figure 5.14: Delivered Fraction of UDP traffic (95% Confidence Interval)

ground traffic load on a link from 10% to 80%.. The PPBP traffic model is used to generate ingress traffic for each flow. The Hurst parameter is 0.8. The COV is 1.0 throughout all the experiments.

Each end host has installed a TCP server, a TCP client, a UDP server, and a UDP client. For every background traffic load, we simulate 10 experiment settings. In each experiment setting, a TCP client randomly pick a TCP server on any other end host. A TCP client's behavior can be modeled as an ON/OFF process: it connects with the TCP server it has chosen, requests a data transfer of 5M bytes and then wait for the requested bytes from the server; after the client receives all the data it has requested, it remains idle for an exponentially distributed period with mean of 5 seconds; when it wakes up, it starts another request and the above process repeats until the simulation is over. Upon receiving a request, a TCP server uses TCP protocol to transfer requested bytes to the client from which the request comes. In each experiment setting, a UDP server also randomly chooses a UDP client on any other host. A UDP server's behavior can also be modeled as an ON/OFF process: it uses UDP protocol to send a file of 5M bytes at the constant rate of 1Mbps, and after it finishes the transfer, it remains idle for an exponentially distributed period with mean of 5 seconds; after the off period finishes, it sends another 5M bytes at the same rate and the above process repeats until the simulation is over. Every experiment setting is simulated for 1000 seconds.

Figure 5.13 presents the average TCP goodput, which is the number of bytes that have been successfully transferred per second, as a function of average background traffic load on a link. The average relative error on this metric over all the background traffic loads is 3.9% with standard deviation 1.3%. Figure 5.14 gives the successfully delivered fraction of UDP traffic as a function of the average background traffic load on a link. The average relative error on it over all the background traffic loads is 0.6% with standard deviation 0.9%. Apparently, the two background traffic

generation methods achieve excellent agreement on both the statistics. Therefore, the time-stepped coarse-grained traffic simulation technique, when used to generate background traffic in realistic topologies, is able to produce simulation results close to those from pure packet-level network simulators.

5.6 Flow Merging

The simulation results in Section 5.5.3 tell us that as the size of the network under analysis grows, its simulation requires more computation and memory space. From Table 5.11, we have seen that simulation of Top-4 cannot be accomplished on the given hardware because of its memory limitation. In this section, we introduce the *flow merging* technique that can help reduce both memory usage and execution time.

The key idea of the flow merging technique is that if multiple flows traversing through an output port are destined for the same egress node, they can be merged into an aggregate flow. Such aggregation, apparently, does not affect the accomplishment of the two objectives discussed in Section 5.3 because both of them consider only the aggregate traffic behavior. We use the feed-forward network in Figure 4.4 as an example. The new network with merged flows is illustrated in Figure 5.15. Flows f_1 and f_2 leave the system at the same place, so they merge into aggregate flow $f'_{1,2}$ before entering port A. Similarly, flows f_3 and f_4 are merged into aggregate flow $f'_{3,4}$ before entering port A, flows f_5 and f_6 are merged into aggregate flow $f'_{5,6}$ before entering port B, and flows f_7 and f_8 are merged into aggregate flow $f'_{7,8}$ before entering port C. Aggregate flow can further be merged with other flows to form a larger flow. For example, aggregate flow $f'_{5,6}$ merges with flows f_9 and f_10 into $f'_{5,6,9,10}$ before entering port D.

After multiple flows are merged at an output port, only the aggregate flow departs from the port. Therefore, the flow merging technique reduces the memory required to maintain the flow states. On the other hand, note that the performance of our algorithm is contingent on how many flow variables need to be settled. The flow merging technique is also helpful in reducing the simulation execution time. In the following discussion, we quantify the reduction on memory usage offered by the flow merging technique. Recall that in Section 5.4.3, we have described how to construct the condensed graph given a network topology and the flows traversing it.

Theorem 7 *Let a network have N nodes, K POPs, and a flow from every POP to every other POP. If the average number of nodes on a flow's path is M and the number of edges in the condensed graph is L, then the ratio of the memory space needed for keeping flow states when the flow merging technique is applied to that when it is not is*

$$O(min\{1, \frac{N}{K \cdot M}\}) \text{ and } \Omega(\frac{L}{K^2 \cdot M}). \tag{5.21}$$

Proof: Let C be the memory space needed to keep a flow's state on a node. Note that in total there are $K(K-1)$ flows in the network if they are not merged. If the flow merging technique is implemented as described, on any node at most one merged flow state is maintained for each

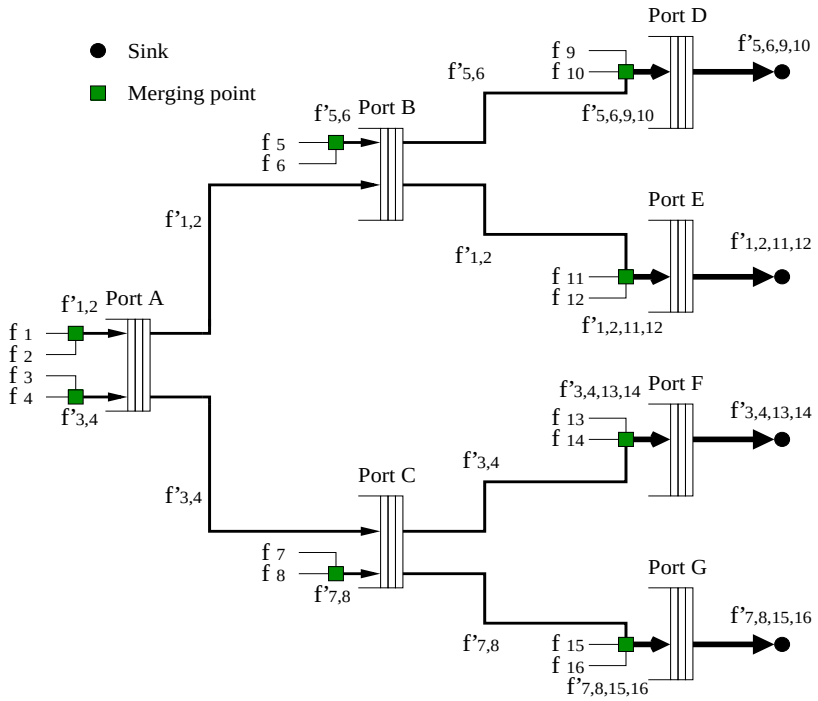


Figure 5.15: Flows Merged in the Feed-Forward Network Shown in Figure 4.4

destination POP; hence, the overall memory space required for keeping flow states is at most $K \cdot N \cdot C$. On the other hand, if the flow merging technique is not applied, the overall memory space needed to keep flow states is $K(K-1) \cdot M \cdot C$. Also note that the memory space needed for keeping flow states when the flow merging technique is applied should not exceed that when it is not. Therefore, the ratio of the memory space needed for keeping flow states when the flow merging technique is applied to that when it is not is $O(min\{1, \frac{N}{K \cdot M}\})$.

Recall that if there is an edge from port q to port p in the condensed graph, there must exist a flow that traverses ports q and p consecutively. Hence, when the flow merging technique is applied, if there is an edge starting from a port, at least one flow state is maintained at that port. Because there are L edges in the condensed graph, the total number of flow states is then at least $L \cdot C$ in the flow merging case. Therefore, the ratio of the memory space needed for keeping flow states when the flow merging technique is applied to that when it is not is $\Omega(\frac{L}{K^2 \cdot M})$. □

In some POP-level topologies, every node is a POP. From Theorem 7, we have the following corollary:

Corollary 8 *Let a POP-level topology have K POPs and a flow from every POP to every other POP. If the average number of nodes on a flow's path is M and the number of edges in the condensed graph is L, the ratio of the memory space needed for keeping flow states when the flow merging technique is applied to that when it is not is $O(1/M)$ and $\Omega(\frac{L}{K^2 \cdot M})$.*

Topology	COV = 5		COV = 0.5	
	secs/round (20% link util)	secs/round (50% link util)	secs/round (20% link util)	secs/round (50% link util)
Top-1	0.0020	0.0020	0.0031	0.0032
Top-2	0.049	0.049	0.1093	0.1175
Top-3	0.249	0.256	0.5064	0.5997
Top-4	0.755	0.787	1.5437	1.7789

Table 5.12: Average Execution Time per Time-Step Using Flow Merging Technique

Using the flow merging technique, we re-simulate the four topologies in Table 5.10. The average execution times per time-step are given in Table 5.12. Comparing these results with those in Table 5.11, we can see that the flow merging technique reduces the simulation execution times under all settings. Further observations are made as follows:

- The performance gain from the flow merging technique with a high COV is less impressive than that with a low COV. As mentioned before, a high COV may cause relatively high peak rates followed by long "silent" periods. Because in our implementation, the flows with ingress rate 0 are ignored in the computation, whether they are merged on their paths does not affect the performance.

- The flow merging technique achieves higher relative speedup when the traffic load in the network is heavier. Increasing traffic load in the network causes more congested ports and thus more flow variables on the dependence graph in Phase III. Because the flow merging technique reduces the total number of flow variables in the topology, in a network with heavier traffic load, fewer flow variables appear on the dependence graph in Phase III and less computation is thus done in the fixed point iterations.

- The flow merging technique achieves higher relative speedup when a larger topology is simulated. Note that when the network size grows, the average number of flows traversing through a port also increases. If more flows traverse a port, it is more likely that flows destined for the same destination can be merged to form larger aggregate flows and thus the flow merging technique can work more effectively.

- Without the flow merging technique, we are not able to simulate Top-4 when COV is 0.5 on the given hardware because of its memory limitation. The flow merging technique, however,

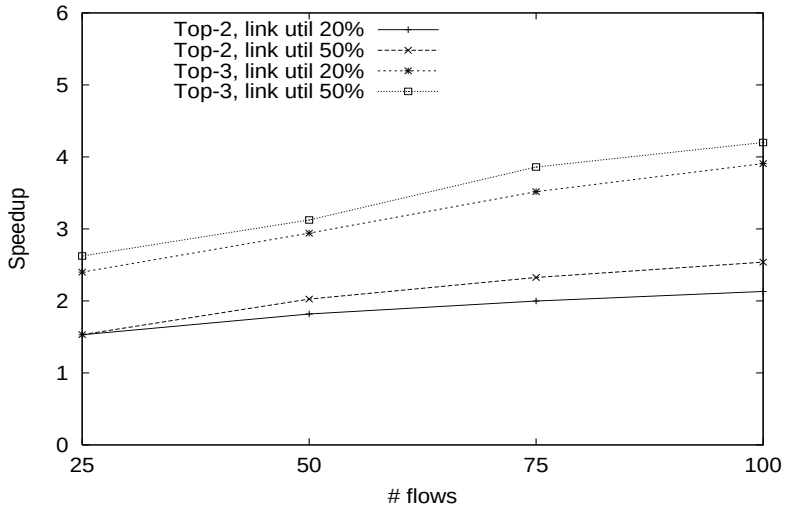


Figure 5.16: Execution Speedup of Simulation with Flow Merging over Simulation without Flow Merging

not only makes it possible to simulate this topology on the given machine, but also simulates it in real time under both traffic loads. Recall that the time step is 5 seconds in the experiments.

In order to gain further understanding on how much performance improvement we can obtain from the flow merging technique, we have designed another set of experiments. We use the same topologies in Top-2 and Top-3 but vary the number of flows that a node directs to other nodes. In the experiments, every router in the new topologies directs k flows to the routers in its own backbone and k flows to the routers in each other backbone, where k is varied between 25, 50, 75, and 100. The PPBP traffic model is still used to generate ingress rates for each flow. The COV is 0.5 and the Hurst parameter is 0.8. We simulate two traffic loads, one with average link utilization 20% and the other with average link utilization 50%. Figure 5.16 gives the relative speedup of the simulation with the flow merging technique over the simulation without it. From the results, we can see that simulation speedup obtained from the flow merging technique is approximately linear with the number of flows that a node directs to other nodes. It also confirms the two observations made earlier: we can achieve better performance gain from the flow merging technique by increasing either the traffic load in the network or the topology size.

5.7 Parallel Algorithm

The sequential algorithm we have described in Section 5.5 offers the capability of simulating a reasonably large network on a uni-processor. However, when the size of the network continues to grow, the limited computation and memory resources possessed by a uni-processor will ultimately become the bottleneck. It is natural, therefore, to parallelize the sequential algorithm on a distributed

memory multiprocessor. We can then combine both advantages of high abstraction-level models and high-performance computing to improve the performance of large-scale network simulation. Some new notations are introduced in this section. Let P be the collection of processors used in the parallel simulation. The processors in P are denoted as p_1, p_2, ..., and $p_{|P|}$, where $|P|$ is the total number of processors.

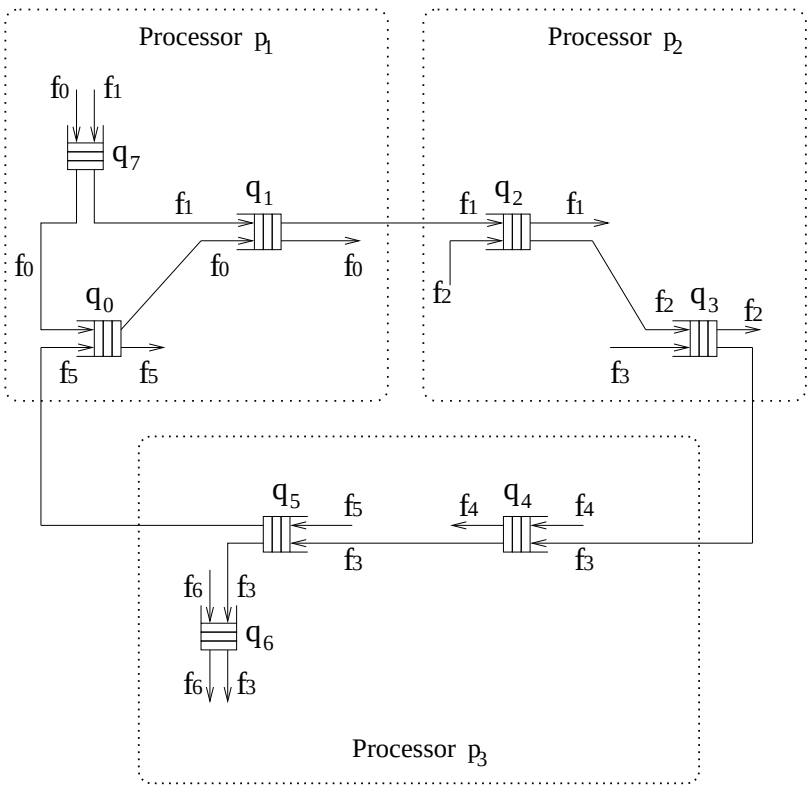


Figure 5.17: Partitions of the Topology in Figure 5.5

The example shown in Figure 5.5 is also used to illustrate how we parallelize each phase in the sequential algorithm. Suppose that the original topology is divided into three partitions, each simulated by a single processor. The partition is shown in Figure 5.17. Flow f_1's path is on both processors, p_1 and p_2; flow f_3's path is on both processors p_2 and p_3; flow f_5's path is on both processors p_3 and p_1.

In the following subsections, we first describe how to parallelize the three phases in the sequential algorithm, especially the synchronization protocol among processors. After that, we investigate the scalability of the parallel algorithm under both fixed and scaled problem sizes.

5.7.1 Algorithm Description

The parallel algorithm keeps the basic structure of the sequential algorithm. Now we discuss how to parallelize each phase in the sequential algorithm.

Phase I: Rule-Based Flow Update Computation. At the beginning of every time step, there is no necessity for all the processors to synchronize with each other. Each processor starts by applying the prioritized rules on the ports maintained locally. As in the sequential algorithm, after a processor applies a rule on a flow traversing through a local output port, it updates its state at the next output port. However, it is possible that the next output port is located on another processor. We use *flow update* messages to deliver such flow update information. For example, after resolving port q_1 on processor p_1, the flow variable λ_{f_1,q_2} can be settled. Since q_2 is located on processor p_2, a message carrying the settled rate of λ_{f_1,q_2} is sent from processor p_1 to processor p_2. Every processor keeps monitoring arriving flow update messages. When such a message is received, the processor processes the message according to the specified action. In the example, when processor p_2 receives the flow update message from processor p_1, it immediately settles the flow variable λ_{f_1,q_2} with the rate carried in the message.

At some point, a processor may find that no local port satisfies the pre-condition to any rule. It should not leave Phase I unless it ensures that no flow update messages will come from other partitions in the current time step. Ignoring such messages may cause the following problems when a processor enters Phase II:

- a flow variable is not settled even though the upstream port on the flow's path has already been resolved;
- a flow variable is not settled even though the upstream port on the flow's path has been identified as transparent and the corresponding input flow variable at that port has been settled;
- a port is in the *unresolved* state even though a tighter upper bound on an input flow variable established by the upstream port on the flow's path can make it transparent;
- a flow variable is still in the *unsettled* state.

These issues can lead to more flow variables that appear on the dependence graph in Phase III. It is necessary, therefore, to ensure that when a processor enters Phase II, all other processors have also finished Phase I. This can be done with *barrier synchronization* that is widely used in parallel computation. A standard barrier primitive is a function that blocks the execution of the calling processor until all other participating processors also invoke the function, at which time all processors are allowed to continue. As to our problem, in order to ensure that a processor can safely enter Phase II, a barrier should be invoked after it finishes all the work in Phase I. However, a standard barrier algorithm described as above is inappropriate here for the following two reasons. First, if a processor, after finding that no local ports satisfy the precondition to any rule, decides to enter the barrier, it cannot process any flow update message that arrives later. Second, as far as our problem is concerned, it is difficult for a processor to determine whether a new flow update message

will arrive later. In the example topology shown in Figure 5.17, suppose that processor p_1 receives a message from processor p_3 that bounds the rate of flow variable $\lambda^{(in)}_{f_5,q_0}$. At this point, it cannot determine whether it is safe to enter the barrier because it is likely that processor p_3 will send it another flow update message in the future (e.g., one that sets a tighter upper bound on $\lambda^{(in)}_{f_5,q_0}$ or one that settles its rate).

Such dilemma is solved with non-committal barrier synchronization [103]. In contrast to a standard barrier, non-committal barrier makes it unnecessary for a processor to enter the barrier with the guarantee that all its pre-synchronization computation must have been completed. Rather, a processor invoking a non-committal barrier is allowed to further receive computation messages if only it is still in the barrier. In the non-comittal barrier algorithm, every processor maintains both counts on the computation messages it has sent and received. With aid of a tree structure, each processor keeps a set of neighbor processors in different *dimensions*. A dimension can be viewed as a level in the auxiliary tree. Each processor exchanges the counts on computation messages sent and received with its neighbor processors from the lowest dimension to the highest dimension; it is able to progress past the barrier only after it has agreed on the counts on the computation messages sent and received with its neighbor processors in all the dimensions. A processor may receive a computation message while it is checking the counts on the computations messages sent and received with the neighbor processor in a dimension; when this occur, the algorithm rolls back to the lowest dimension. This idea is similar to the optimistic synchronization protocol discussed in Section 2.4.1. The non-committal barrier algorithm requires $O(log^2|P|)$ space on each processor, and in the absence of rollbacks, requires $O(log^2|P|)$ parallel execution time (recall that $|P|$ is the number of participating processors).

Non-committal barrier synchronization solves our problem by allowing processors to process the computation messages received after it has invoked a non-committal barrier primitive. However, if rollbacks occur frequently in the algorithm, its performance may deteriorate significantly. When a processor rolls back to the lowest dimension, it has to exchange again the counts on the computation messages sent and received with all its neighbors. This process itself requires communication messages to exchange synchronization information. Hence, if rollbacks occur frequently, a significant part of computation is consumed on processing the synchronization messages. In short, the optimistic feature of non-committal barriers provides high flexibility for the processors to synchronize with each other, but their performance is affected by how frequently rollbacks happen.

In order to avoid unnecessary rollbacks after premature entrance to the non-committal barrier, we let Phase I invoke the barrier primitive after it finishes as much pre-synchronization computation as possible. The following two conditions must be satisfied before a processor enters a non-committal barrier in Phase I:

(1) no ports on the local partition satisfy the pre-condition to any of the 6 rules described in Tables 5.1-5.6;

(2) no flow variable on the local partition is in the *unsettled* state.

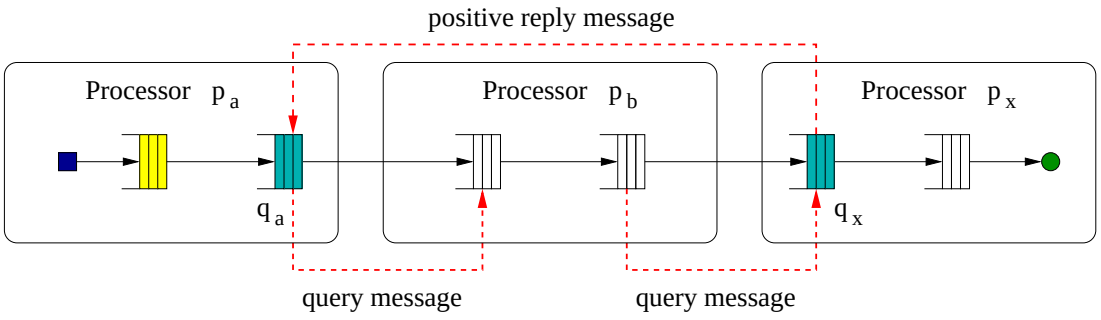
Condition (1) is evident because if there exists a port satisfying the pre-condition to any of the 6 rules,, we can simply apply that rule on it. Condition (2) is based on Lemma 8. From that lemma,

we know that every flow variable must be in the *settled* or *bounded* state after Phase I. Hence, if there exists a flow variable still in the *unsettled* state in Phase I, a flow update message must arrive later from another processor that updates its state to either *settled* or *bounded*. In order to avoid the unnecessary rollback in the non-committal barrier when that occurs, the processor postpones entering the barrier until that message is received.

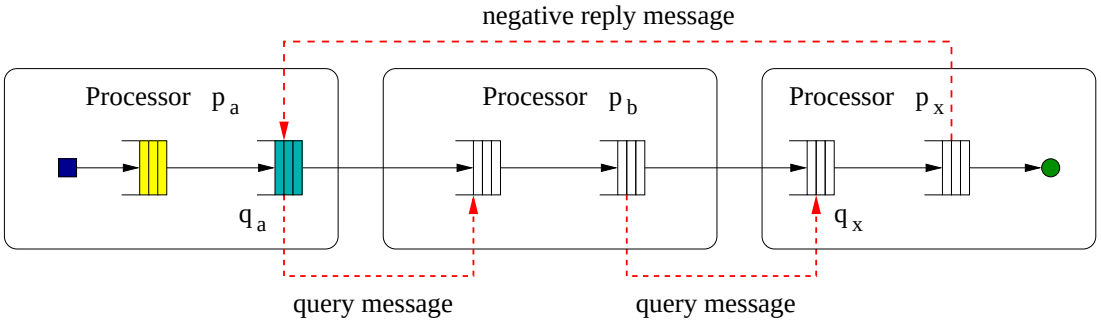
Phase II: Dependence Graph Generation. Recall that in the sequential algorithm, this phase is divided into two steps. In the first step, the dependence graph is created by putting every port in the *unresolved* state into its vertex set and then adding directional edges that reflect the dependence relationship among the unsettled flow variables. In the second step, all the vertices (or ports) with outdegree 0 are iteratively pruned from the dependence graph.

On a distributed computing platform, the ports in the *unresolved* state can be located on multiple partitions, each of which then has only partial knowledge of the whole dependence graph. Consider that processor p_a processes port q_a in Phase II. For every input flow variable $\lambda^{(in)}_{f,q_a}$ ($f \in F_q$) in the *unsettled* state, the local processor needs to find the next port q_x on flow f's path that is also in the *unresolved* state (if such q_x exists). Port q_x, however, may not be found on the local partition. When this occurs, the local processor sends a *query* message to the processor to which flow f goes after departing from processor p_a. Let p_b denote the processor that receives this *query* message. Processor p_b, after receiving the message, continues searching port q_x on the piece of flow f's path that it keeps. If port q_x is found on processor p_b, then processor p_b sends a *positive reply* message to processor p_a and adds a *backward* inter-processor edge from $< p_a, q_a >$ to $< p_b, q_x >$ to the part of dependence graph that is maintained locally; if q_x is not found and flow f goes to a sink on processor p_b, then a *negative reply* message is sent back to processor p_a; if q_x is not found on processor p_b and flow f goes to another processor p_c, then processor p_b forwards the *query* message from processor p_a to processor p_c. When processor p_c receives the query message, it processes it in the same way as processor p_b. Finally, if processor p_a receives a *positive reply* message from, say processor p_x, it adds a *forward* inter-processor edge from $< p_a, q_a >$ to $< p_x, q_x >$ to the part of the dependence graph that is maintained locally; if processor p_a receives a *negative reply* message, it simply ignores it.

In the sequential algorithm, the two steps in this Phase can be finished in strict time order. In the parallel algorithm, however, serializing these two steps requires extra synchronization after the first step. In order to avoid such synchronization cost, we combine these two steps in the parallel algorithm. Every port in the dependence graph keeps counts on both how many *query* messages have been sent and how many *reply* messages have been received. When the two counts at a port match at some point, we know all *query* messages sent out from this port have been replied; in this way, we are able to determine whether its out-degree is 0: a port with outdegree 0 must have neither intra-processor edges pointing to it nor *forward* inter-processor edges. After a port is identified as having outdegree 0, it is specially marked as *detached* and also removed from the vertex set of the dependence graph; all the intra-process edges pointing to it is removed from the partial dependence graph maintained locally. Some *forward* inter-processor edges on other processors may still point to this port. For each of such edges, an *edge-removing* message is sent to the port at the other end. When that port receives this message, it removes the corresponding *forward* inter-processor



(1) Positive reply message



(2) Negative reply message

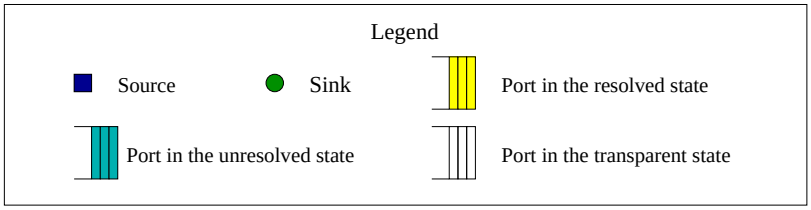


Figure 5.18: Dependence Graph Generation across Multiple Processors

edge and also checks whether its own out-degree decreases to 0; if so, it repeats the above process. In addition, owing to asynchrony, a *query* message may arrive at a port that is marked as *detached*. When this occurs, a *negative reply* message is simply sent to the port that initiates the *query* message. We illustrate messages communicating between processors in Figure 5.18.

All the processors are synchronized before they proceed into Phase III. A similar problem arises because it is difficult for a processor to ensure that no more messages will arrive in the future. Unless a processor has no queues with *forward* inter-processor edges, some *edge-removing* messages may arrive later. Hence, non-committal barrier synchronization is also used here before a processor decides to proceed into the next phase.

In Phase I, we mentioned that frequent rollbacks in a non-committal barrier will negatively affect its performance. For the same reason, we let a processor invoke a non-committal barrier primitive only after it finishes as much pre-synchronization computation as possible. More specifically, a processor must satisfy the following three conditions before it is allowed to enter the barrier:

(1) no port in the partial dependence graph locally maintained has out-degree 0;

(2) for every *query* message sent out from a port, an either *positive* or *negative reply* message corresponding to it must have been received;

(3) for every flow f that traverses a sequence of ports in the partial dependence graph maintained on the local partition, denoted by $(q_1, ..., q_k)$, if flow variable $\lambda_{f,q_1}^{(in)}$ is not in the *settled* state, then: (a) there exists an intra-processor edge between any two successive ports in sequence $(q_1, ..., q_k)$; (b) if flow f comes from another processor, then a *query* message for this flow must have been received at port q_1; (c) if flow f goes to another processor, then a *query* message for this flow must have been sent from port p_k to that processor.

Condition (1) is directly derived from Lemma 9. Condition (2) states that the counts on *query* and *reply* messages at a port must match. Case (a) in Condition (3) ensures that all the intra-processor edges must have been added before the processor enters the barrier; Case (b) in Condition (3) states that if no variable corresponding to a flow is in the *settled* state at any port in the dependence graph maintained locally, this flow must come from another processor and thus a *query* message is expected; Case (c) in Condition (3) means that if a flow leaves for another processor with an unsettled rate, a *query* message should be sent to it for this flow. Prediction on message arrivals enables a processor to postpone entering the non-committal barrier until necessary.

Phase III: Fixed Point Iterations and Residual Flow Update Computation. In this phase, the fixed point solution is applied to determine the remaining unsettled flow variables which are associated with the ports in the dependence graph. We have considered the following two methods to implement the fixed point iterations on a distributed memory multiprocessor:

- **Migration-based approach:** We migrate the whole dependence graph onto the same memory space, apply the fixed point solution to determine the unsettled flow variables, and when the iterations converge or the number of iterations reaches the upper bound, migrate the settled flow variables to their original memory space.

- **Message-passing-based approach:** In every iteration in the fixed point solution, we use communication messages to deliver updated estimates on the unsettled flow variables associated with the ports in the dependence graph.

Migration-based approach produces a relatively small number of communication messages compared with message-passing-based approach. In a computation architecture where communication bandwidth is the performance bottleneck, this approach is an appealing solution. The scalability of this approach is, however, very limited. As the problem size grows, the size of the dependence graph may also increase. Hence, it is possible that the dependence graph has too many ports in it so that a single memory space cannot accommodate it. On the other hand, if only a single processor is involved in the fixed point iterations and all the others are waiting for the results, significant load balancing problem results. In our implementation, therefore, the second approach is adopted.

Now we discuss some details on the message-passing-based method. Two specific questions regarding it needs to be answered. One is *how a processor knows it should start a new iteration*, and the other is *how a processor knows it should terminate the fixed point iterations when the termination criterion is satisfied*. As for the first question, a processor knows exactly how many local flow variables not in the *settled* state from the partial dependence graph it maintains. For the sake of explanation, we use $\vec{\boldsymbol{\lambda}}_g(p)$ to denote the vector of unsettled flow variables in the partial dependence graph maintained on processor p. After processor p finishes an iteration, it updates some estimates on flow variables in $\vec{\boldsymbol{\lambda}}_g(p)$, and also sends out messages to update the estimates of unsettled flow variables in the partial dependence graph maintained on the other processors. Afterwards, the estimates on some flow variables in $\vec{\boldsymbol{\lambda}}_g(p)$ may not have been updated for the next iteration. Under such circumstance, the processor has to wait for messages from other processors to have them updated. It is until the estimates on all the flow variables in $\vec{\boldsymbol{\lambda}}_g(p)$ have been updated that processor p starts the computation for the next iteration.

Concerning the second question, we address it as follows: after finishing an iteration, all the processors exchange the maximum relative difference between the successive estimates on the unsettled flow variables in the partial dependence graph maintained locally. This can be efficiently implemented with a standard *max* reduction operator on a typical distributed memory architecture [111]. If the termination criterion uses other distance norms like the Euclidean distance norm, extra work besides a standard reduction operator may be necessary. Once a processor knows the global maximum relative difference between estimates on unsettled flow variables in the successive iterations, it can decide whether it should terminate the fixed point iterations.

After the fixed point iterations terminate, the processors can resume the flow update computation in Phase I to settle the remaining unsettled flow variables. For the same reason presented in Phase I, we put a non-committal barrier at the end of this phase. Therefore, it is ensured that all computation messages communicated at the current time step have been successfully received by their destinations before it finishes.

5.7.2 Scalability Analysis

In this section, we use simulation experiments to demonstrate the scalability of the parallel algorithm. The performance of a parallel program is relevant to several parameters, including the problem size, processor count, and the architecture on which it is executed. All the experiments discussed in this section are executed on *Tungsten*, a multiprocessor supercomputer at the National Center for Supercomputing Application (NCSA)[2]. A brief technical introduction to *Tungsten* is given as follows. It has 1480 computation nodes, each of which has a *32*-bit 3.2GHz Intel Xeon dual-processor and 3GB memory; the nodes are interconnected with Myrinet 2000. The operating system used on each node is Linux 2.4.20 (Red Hat 9.0).

In the following subsections, we study how the scalability of our parallel program is affected by the processor count and the problem size. Two sets of experiments are discussed, one with fixed problem size and the other with scaled problem size.

5.7.2.1 Scalability with Fixed Problem Size

The purpose of the first set of experiments is to evaluate how effectively the parallel algorithm described in Section 5.7.1 performs with an increased number of processors. We extend the ATT backbone depicted in Figure 4.7 in the experiments. The extended topology contains 32 ATT backbones. Each ATT backbone has 27 nodes, which are numbered from 1 to 27. From the 27 nodes, we choose the three ones with the highest degrees as bordering nodes through which the backbone is interconnected with other backbones. Between every two backbones, we use a link to directly connect pairs of bordering nodes that have common node identifiers. Hence, all the bordering nodes that have the same node identifiers in the topology form a clique. The internal links in each backbone have bandwidth 100Mbps, and the external links that connect two bordering nodes have bandwidth 1000Mbps. The traffic pattern in the experiments is *"all-to-all"*: from each node in the topology, there is a flow directed to every other node. Since there are 864 (i.e., 32×27) nodes in the topology, the total number of flows is 745,632 (i.e., 864×863). The ingress traffic of each flow is modeled with a PPBP traffic model, whose COV (i.e., ratio of standard deviation to mean) is 1 and Hurst parameter is 0.8. We vary the mean of each flow's ingress rate to obtain two traffic intensity levels in the network, one having average link utilization 50% and the other having average link utilization 80%.

The number of processors onto which the topology is partitioned varies between 1, 2, 4, 8, 16, and 32. In most cases, we run the parallel program to execute 200 time steps. When there is a single processor, we execute only 20 time steps for the following reason. In order to isolate the computation cost on generating sample ingress rates for each flow using the PPBP model from the computation cost of our parallel program, we precompute the sample ingress rates for each flow and store them throughout the simulation. However, when a single processor is used, the memory associated with it cannot accommodate the simulation for 200 time steps. Hence, only 20 time steps are simulated in this case. We observe only small variance on the execution time per time step. The accuracy of the simulation results is thus hardly affected.

[2] http://www.ncsa.uiuc.edu/

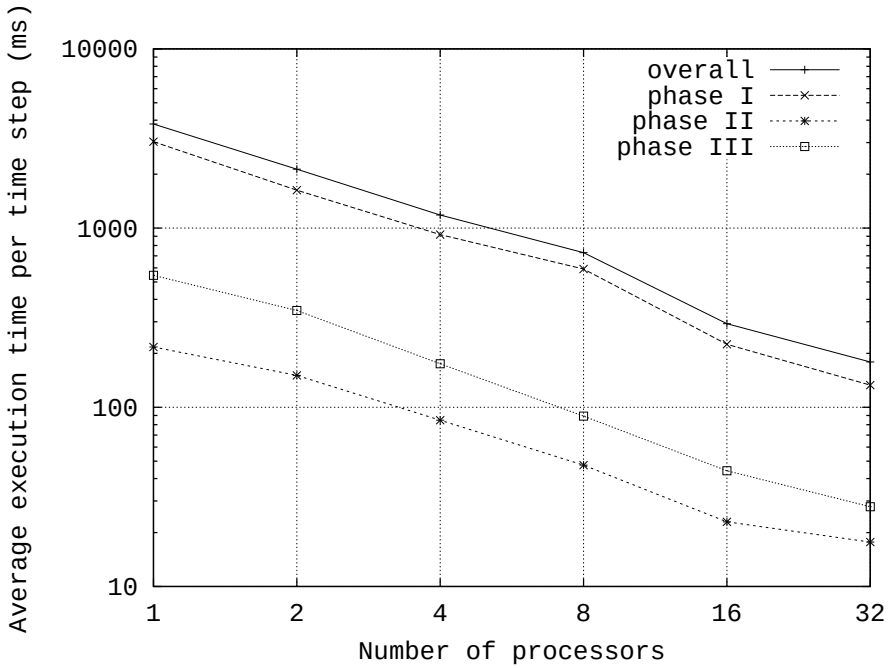
In Figures 5.19 and 5.20, we give the scalability results when the average link utilization is 50% and 80% respectively. Each figure depicts the average execution time and the relative speedup of the parallel algorithm as we vary the number of processors from 1 to 32. From the simulation results, we observe that as the number of processors increases, the average execution time per time step decreases monotonically regardless of the traffic load in the network. This suggests that increasing the number of processors up to 32 is always productive. Note that in some cases it may not be true because as the number of processor grows, the increased communication and synchronization cost may offset the performance gain from parallelizing the computation.

Throughout all the experiments, good efficiency has been observed. The efficiency of a parallel program is defined to be its relative speedup over the corresponding sequential program divided by the number of processors used [37]. A high efficiency means that processors are effectively used in the parallel execution. In our experiments, when the average link utilization is 50%, the mean efficiency from using more than one processor is 0.766, and when the average link utilization is 80%, the mean efficiency from using more than one processor is 0.755. However, we have also observed that the efficiency of our parallel program is not a monotonically decreasing function of the number of processors used in the simulation. It is only a general trend because exceptions happen in some cases. For example, when the average link utilization is 50%, if 8 processors are used, the efficiency is 0.653, but if 16 processors are used, the efficiency is 0.814; when the average link utilization is 80%, if 8 processors are used, the efficiency is 0.660, but if 16 processors are used, the efficiency is 0.774. This is counter-intuitive because usually, when we increase the number of processors, the cost on processing computation and synchronization messages also increases, thus decreasing the efficiency of the parallel program.

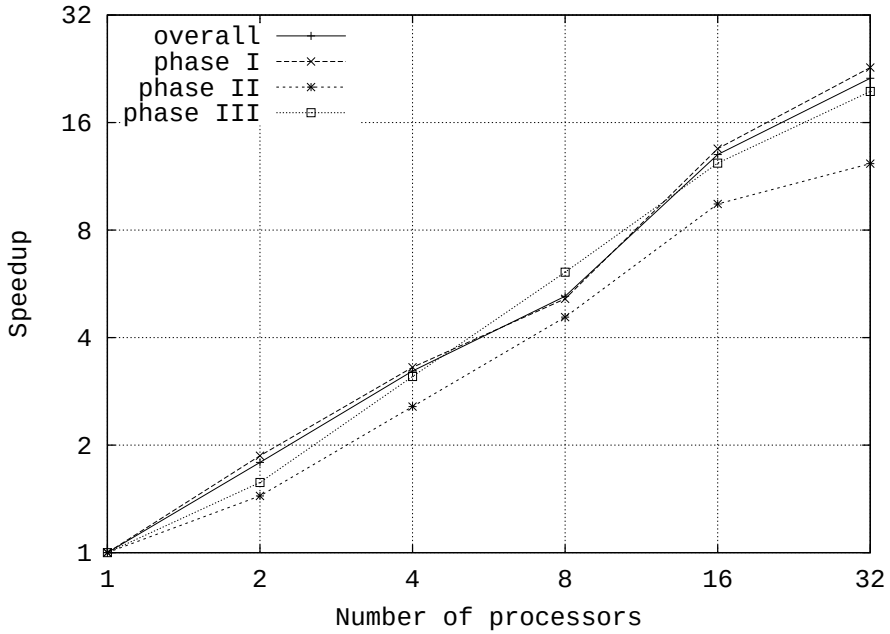
A close examination of the simulation results reveals that the above phenomenon actually originates from the first phase, whose execution time dominates the overall execution time. Note that as the number of processors varies in the simulation, each processor actually behaves differently in the first phase. In our implementation, a processor gives higher priority to processing local ports that are applicable to any of the 6 rules than sending flow update messages to other processors. The heuristics behind this implementation is that a processor is able to aggregate flow update information of many flows into the same physical communication message if they are destined to the same processor. Then, given the same problem, the average size of a computation message when there are fewer partitions is relatively larger than that when there are more partitions. The performance of MPI point-to-point communications, which are used in our implementation, is known to be contingent on the message size. Empirical results show that such performance gap can be significant with different message sizes and the communication throughput is not a monotonic function of the message size[3]. This suggests that the efficiency of our parallel algorithm may not be a monotonically decreasing function of the number of partitions in the simulation, as we have observed from the simulation results.

Table 5.13 presents the proportion of the execution time consumed by each phase to the average execution time per time step. From the results, we can see that given a certain traffic intensity level, the proportion corresponding to each phase alters only slightly as we vary the number of

[3] http://www.mhpcc.edu/training/workshop2/mpi_performance/MAIN.html

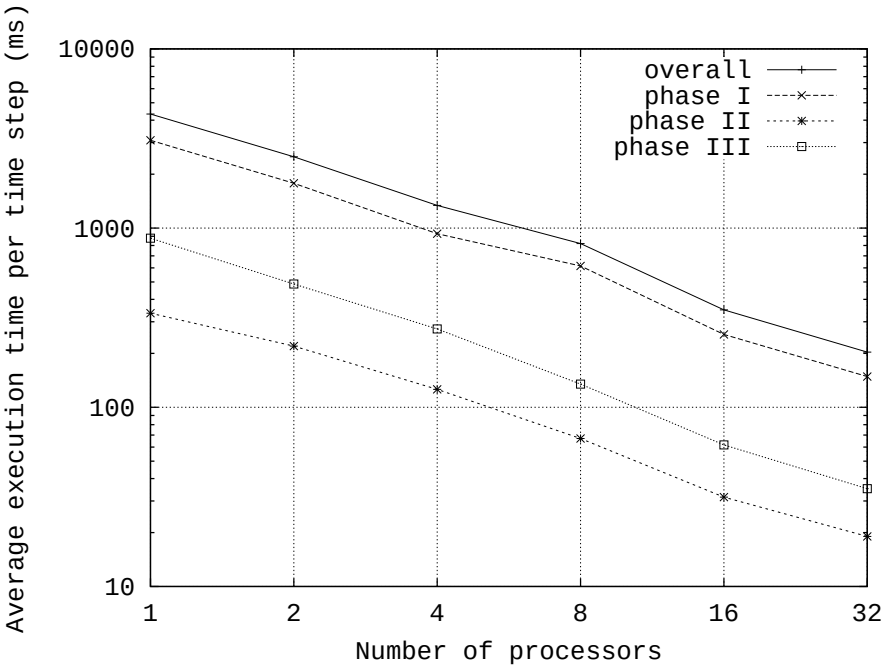


Exeuction Time

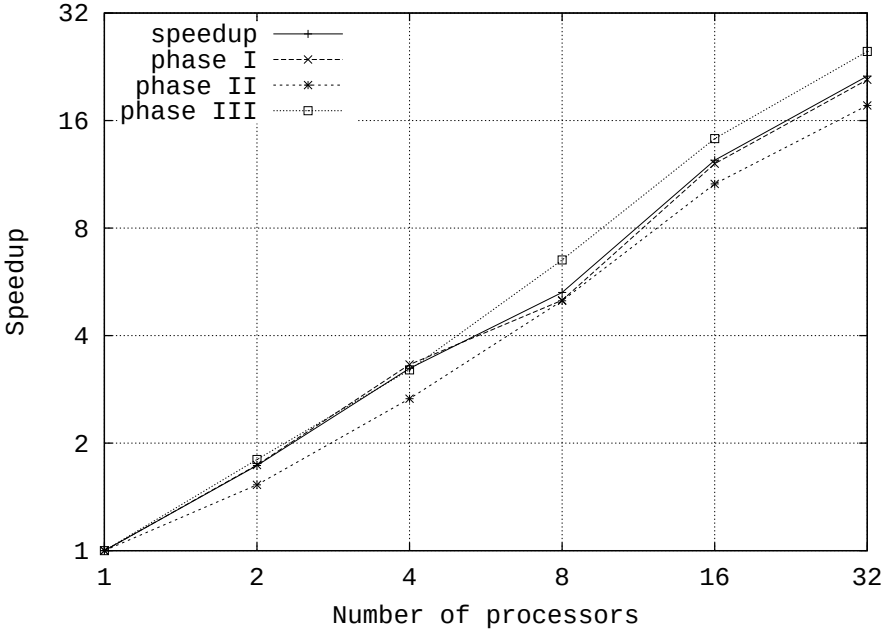


Relative Speedup

Figure 5.19: Scalability Results with Fixed Problem Size (Link Utilization 50%)



Exeuction Time



Relative Speedup

Figure 5.20: Scalability Results with Fixed Problem Size (Link Utilization 80%)

Load	Phase	Timelines					
		1	2	4	8	16	32
50%	I	79.9%	76.6%	78.0%	81.2%	77.0%	74.5%
	II	5.7%	7.1%	7.2%	6.5%	7.9%	9.9%
	III	14.3%	16.3%	14.8%	12.3%	15.2%	15.6%
80%	I	71.8%	71.6%	70.0%	75.3%	73.2%	73.3%
	II	7.8%	8.8%	9.5%	8.2%	9.1%	9.4%
	III	20.4%	19.6%	20.6%	16.5%	17.7%	17.3%

Table 5.13: Proportion of Execution Time Consumed by Each Phase w.r.t the Average Execution Time per Time Step

processors in the simulation. It is also evident that the execution time by Phase I dominates over the other two phases by taking more than 70.0% of the average execution time per time step under both traffic intensity levels. This is explicable in that our algorithm aggressively searches for ports that can be settled in Phase I and thus minimizes the number of ports involved in later fixed point iterations. The number of ports involved in fixed point iterations is approximately 5% and 8% of the total number of ports when the average link utilization is 50% and 80% respectively. Another observation is that as the traffic load in the network increases, the proportion of the execution time taken by Phases II and III to the average execution time per time step also increases. When the traffic intensity level in the network increases, a larger fraction of ports are congested and more computation and communication are thus necessary to form the dependence graph in Phase II and update the estimates of the unsettled flow variables in the dependence graph in Phase III.

5.7.2.2 Scalability with Scaled Problem Size

In the previous section, we discussed how our parallel program performs in a given problem using varied number of processors. When the scalability of a parallel program is analyzed, it is an important objective that increasing the number of processors in the simulation can shorten the execution time. On the other hand, there is another motivation for using large parallel computers, that is, to solve larger problems with more computation resources. It is, therefore, also important that parallel programs are able to sustain its efficiency under scaled problem size. In this section, we use experiments to demonstrate the scalability of our parallel algorithm from this perspective.

In the new experiments, we extend the router-level Exodus backbone, which is listed as Top-2

in Table 5.10. In the simulation that runs on k processors, the topology has k Exodus backbones, each of which is a partition handled by one processor exclusively. The Exodus backbone has 244 routers, whose identifiers are numbered from 1 to 244. From these routers, we select the 17 routers that have the highest degrees in the backbone as its bordering routers. Among the topology with k backbones, every pair of bordering routers with a common identifier are directly connected with an inter-domain link; hence, all the bordering routers with the same identifier form a clique of k nodes. Each link that interconnects two backbones has bandwidth 2,488Mbps. For the sake of explanation, we number the k backbones from 1 to k. The intra-domain traffic pattern is *"all-to-all"*, that is, from each router in any backbone, there is a flow directed to every other router in the same backbone. Hence, there are $59,292 \cdot k$ (i.e., $k \times 243 \times 244$) intra-domain flows. The inter-domain traffic pattern is circular: from each router in the i-th backbone ($1 \leq i \leq k$), there is a flow directed to every router in the j-th backbone, where j is $i+1$ if i is not equal to k or 1 otherwise. Hence, there are $59,536 \cdot k$ (i.e., $k \times 244 \times 244$) inter-domain flows. The total number of flows in the simulation using k processors is $118,828 \cdot k$. As before, the ingress traffic of every flow is modeled with a PPBP traffic model whose COV parameter is 1 and Hurst parameter is 0.8. We vary the mean of the ingress rate of each flow to obtain two traffic intensity levels, one with average link utilization 50% and the other with average link utilization 80%.

In the experiments, we vary the number of processors from 2 to 32. In each experiment, 200 time steps are simulated. The average execution times per time step for both traffic intensity levels are depicted in Figure 5.21. We do not give the results with only one processor because there is no inter-domain traffic and the workload on it thus differs from the average workload per processor when there are more than one processors in the simulation. It is evident that under both traffic intensity levels, the average execution time per time step hardly changes as the problem size scales. When the average link utilization is 50%, the average execution time per time step increases from 780.6 milliseconds with 2 timelines to 794.7 milliseconds with 32 timelines, only by 1.8%; when the average link utilization is 80%, the average execution time per time step increases from 807.9 milliseconds with 2 timelines to 844.6 milliseconds with 32 timelines, only by 4.5%.

We also notice the trend that the average execution time of Phases II and III increases slightly with growing number of processors. However, we do not observe the same trend for the first phase. This is because the rule-based flow update computation, whose execution is contingent on the orders of flow update message arrivals, brings relatively high variation to its performance. This contrasts with the other two phases, in which the workload on each processor is comparatively stable once the states of all the flow variables and all the ports have been determined after the first phase. Hence, the performance of the last two phases is more significantly affected by the synchronization cost, which increases with a growing number of processors.

In Figure 5.22, we give the average count on the messages sent by each processor under two traffic intensity levels. The curves for the synchronization message counts in Phases II and III are so close that they overlap with each other in the both graphs. It is clear that the count on the synchronization messages sent by each processor grows with increased number of processors. Recall that processors in Phase I synchronize with each other using non-committal barriers before they enter Phase II. The key idea of non-committal barrier synchronization is that processors reach a consensus on counts on computation messages sent and received. Without any rollback, a processor has to

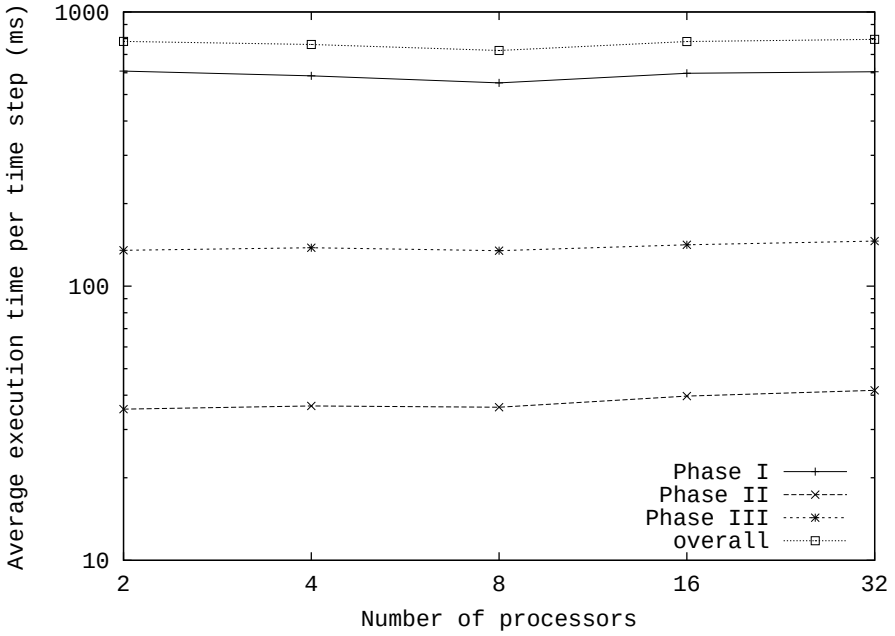
synchronize with $log|P|$ other processors, where $|P|$ is the total number of processors. Therefore, as more processors are used, each processor has to synchronize with more other processors, increasing the number of synchronization messages. However, we have also observed an exception when the average link utilization is 50%: the count on the synchronization message sent in Phase I under 8 processors is smaller than that under 4 processors. Note that there are more computation messages sent by each processor under 4 processors than under 8 processors. As the number of computation messages grows, rollbacks can occur more frequently in the non-committal barrier, and therefore, more synchronization messages are generated.

Concerning the computation messages, we have observed that the numbers of computation messages sent in both Phases II and III remain almost unaltered as the number of processors varies. Recall that the traffic pattern a processor observes in the simulation is the same across all the cases with different number of processors. Therefore, the distribution of congested ports exhibits a similar structure across timelines, regardless of the number of timelines the topology is partitioned into. Once the congested ports have been identified in Phase I, the work done in Phases II and III bears close resemblance across partitions, each though the number of partitions varies in the experiments. This explains the small variation on the number of computation messages send in Phases II and III.

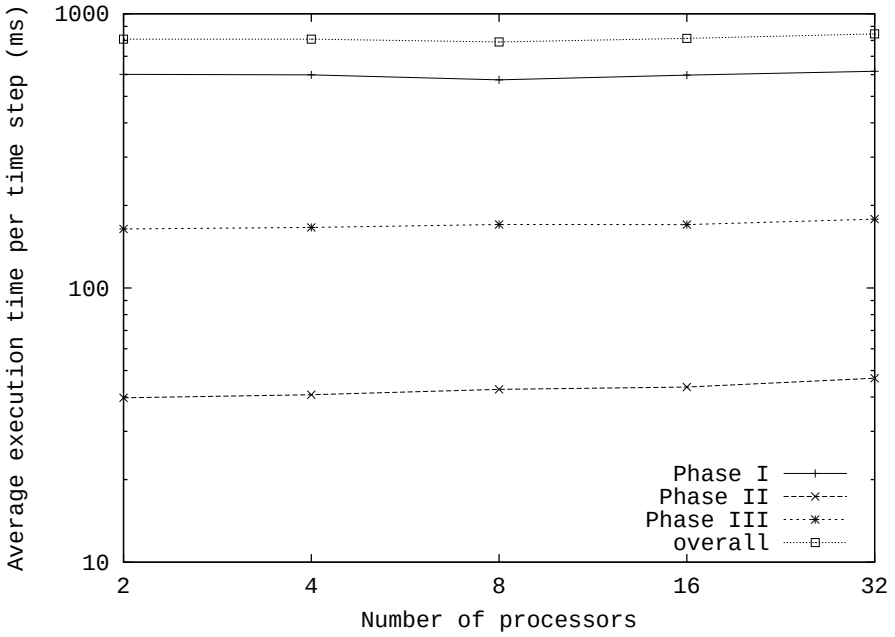
It has also been noticed that the number of computation messages in Phase I decreases as the number of processors increases in the simulation. As we mentioned before, a processor's behavior in Phase I is relatively more dynamic compared with in Phases II and III, because it depends on the order in which flow update messages are received. When we increase the number of processors in the simulation, the total number of messages, including both computation messages and synchronization messages, grows dramatically. Therefore, when there are only a small number of processors, computation messages are delivered to their destinations within a shorter time than when there are a large number of processors. In Phase I, after a processor receives a flow update message, it first updates the corresponding flow variables indicated by the message; after that, it processes the ports that satisfy the precondition to any of the 6 rules; when no local ports can be processed, it sends out flow update messages to other processors. If the communication throughput is lower, the above process happens less frequently and relatively more flow variables are packed in the same physical communication messages and thus fewer computation messages are generated in the simulation. This explains the overall trend that an increased number of processors generates decreased number of computation messages per timeline in Phase I, regardless of the traffic intensity levels in the network.

From the simulation results, it can be safely concluded that our parallel program achieves excellent scalability as the problem size scales up to 32 processors. Note that in all the experiments, the average execution time per time step takes less than 1 second. As we have discussed in Section 5.1, the time-stepped fluid-oriented traffic simulation is suited for large-scale background traffic simulation at coarse time scales. In many cases, the time step is configured at orders of seconds or even minutes. Hence, our parallel algorithm opens the avenue of real-time simulation of ultra-large network topologies, which are exemplified by one with 32 tier-1 backbones in our experiments. However, we have also realized some other important problems towards this objective. For example, in our experiments, we assume that the partition is identical in terms of both topology and traffic pattern across all the participating processors. It is imaginable that given a more realistic topology

and traffic pattern, it may not be an easy task to achieve perfect load balancing across partitions. Therefore, an important problem is how to achieve ideal load balancing under realistic topology and traffic. There has been some work in this field [83][84], but a general solution that works well under varied simulation configurations still remains as an open problem. Furthermore, an ideal partitioning algorithm should be aware of the traffic representations. For example, the computation workload for fluid-oriented models may be different from packet-oriented models when they are used to represent the same traffic. In order for our parallel algorithm to perform efficiently, it is important to find a partitioning algorithm that takes its execution property into consideration. In any event, the simulation results demonstrated in this section offer great hope that simulation of ultra-large realistic networks can be accomplished in real time.

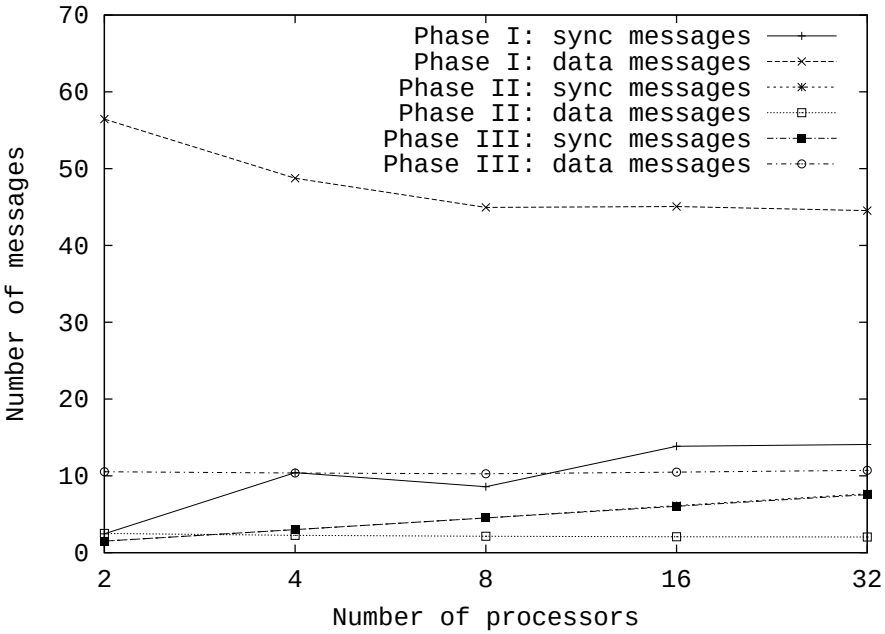


50% Link Utilization

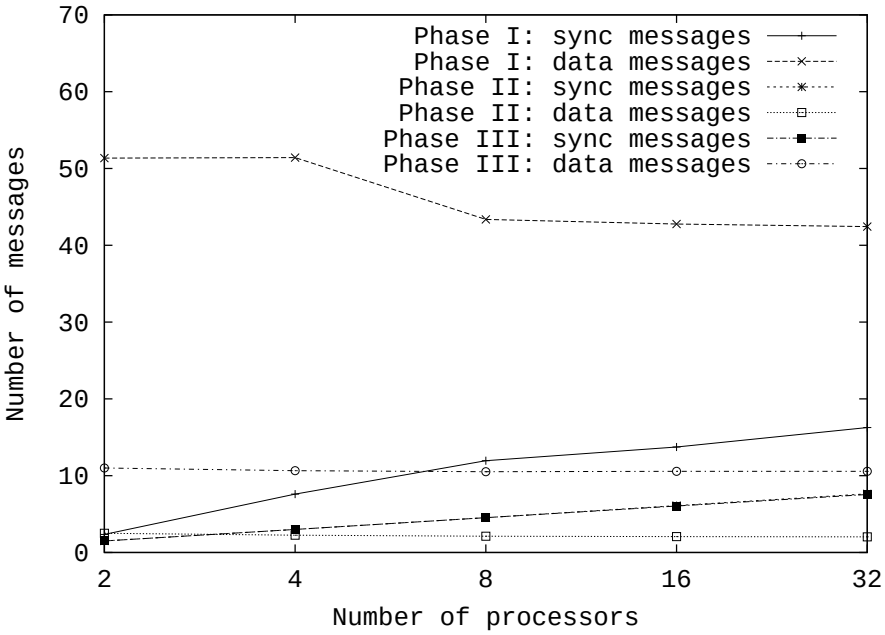


80% Link Utilization

Figure 5.21: Average Execution Time with Scaled Problem Size



50% Link Utilization



80% Link Utilization

Figure 5.22: Message Counts per Processor with Scaled Problem Size

5.8 Rule Modifications for Multi-Class Networks

The rules presented in Tables 5.1-5.6 are suited for networks that have deployed the FIFO scheduling policy. In networks where there are multiple classes of traffic, the scheduling policy in ports may not conform to the FIFO service discipline. However, with some modifications on the rules, the main body of the algorithm can still be used. In this section, we discuss how to change the rules for ports that implement GPS scheduling policy [113]. GPS is a work-conserving scheduling discipline that provides guaranteed service rate for each traffic class, but if some traffic classes do not consume their guaranteed service rates, allows the residual service rates to be shared by other traffic classes. We use the same notations in Section 5.3. We consider a simple case in which there is only one flow in each traffic class. Suppose that n flows traverse port q with GPS scheduling policy. These flows are denoted by f_1, f_2, ..., f_n respectively. Associated with flow i ($1 \leq i \leq n$) is a non-negative weight, ϕ_i, which specifies the guaranteed service rate for flow i

$$\lambda_i^{(g)} = \frac{\phi_i}{\sum_{j=1}^{n} \phi_j} \mu_q. \tag{5.22}$$

Recall that μ_q is the service rate of port q. The guaranteed service rate for each flow provides the service isolation among traffic classes. After distributing the guaranteed service rate to each flow, there may still exist some excess service rate. It is reallocated among the backlogged flows in proportion to their weights. With the GPS scheduling discipline, if flow i is backlogged in the system, then

$$\frac{\lambda_{f_i,q}^{(out)}}{\lambda_{f_j,q}^{(out)}} \geq \frac{\phi_i}{\phi_j}, \quad j = 1, 2, ..., N. \tag{5.23}$$

Note that the departure rate is actually the service rate received by a flow.

The GPS scheduling discipline can be implemented as in Table 5.14. Residual bandwidth C is initialized to be μ_q, the bandwidth associated with port q. Set U consists of flows whose departure rates have not been determined (lines 1 and 2). U is initialized to contain all the flows from f_1 to f_n. The departure rate of each flow is initialized to be its guaranteed service rate (lines 3 and 4). Lines 5 to 9 determine the flows whose traffic can all be served, allocate bandwidth to these flows, and then compute the residual bandwidth. Line 10 removes those flows whose departure rates are determined from set U. If all the departure rates have been determined, the algorithm ends (line 11). Otherwise, the residual bandwidth is allocated to the flows whose departure rates are still unsettled according to their bandwidth share weights. After the allocation, it is possible that some flows receive more bandwidth than their arrival rates. Then, the process repeats itself until all the departure rates are determined. It is proven in [73] that the algorithm realizes the GPS scheduling policy. In the following discussion, we call this the *GPS algorithm*.

Given the GPS scheduling discipline, we discuss how to modify the rules in the algorithm presented in Section 5.5.1. Rule 1 in Table 5.1 states that for a port in the *unresolved* state, if all its input flow variables are settled, the departure rates can be determined according to the FIFO policy. When the GPS scheduling discipline applies, we only need to change the way in which the

1. $C \leftarrow \mu_q$;

2. $U \leftarrow \{f_i\}_{1 \leq i \leq n}$;

3. for every f_j in U

4. $\quad \lambda_{f_j,q}^{(out)} \leftarrow C \cdot \phi_j / \sum_{f_i \in U} \phi_i$;

5. $S \leftarrow \{f_j \mid f_j \in U$ and $\lambda_{f_j,q}^{(in)} < C \cdot \phi_j / \sum_{f_i \in U} \phi_i\}$;

6. if S is empty, the algorithm finishes;

7. for every f_j in S

8. $\quad \lambda_{f_j,q}^{(out)} \leftarrow \lambda_{f_j,q}^{(in)}$;

9. $\quad C \leftarrow C - \lambda_{f_j,q}^{(in)}$;

10. $U \leftarrow U - S$;

11. if U is empty, the algorithm finishes;

12. for every f_j in U

13. $\quad \lambda_{f_j,q}^{(out)} \leftarrow C \cdot \phi_j / \sum_{f_i \in U} \phi_i$;

14. goto line 5;

Table 5.14: Implementation of GPS Scheduling Policy

Precondition	$\exists q \in Q\, s.t.$ $S(q) = unresolved \wedge$ $\exists f_j \in F_q(t_k)\, s.t.$ $S(\lambda^{(in)}_{f_j,q}(t_k) = settled \wedge$ $\lambda^{(in)}_{f_j,q}(t_k) \leq \mu_q \cdot \phi_j / \sum_{f_i \in F_q(t_k)} \phi_i$
Action	$S[\lambda_{f_j,\Pi(f_j,q)}(t_k)] \leftarrow settled$ $\lambda^{(in)}_{f_j,\Pi(f_j,q)}(t_k) \leftarrow \lambda^{(in)}_{f_j,q}(t_k)$

Table 5.15: Rule 2'

departure rates are computed from Equation (5.3) to the GPS algorithm. The modified rule is called Rule 1'.

Because the GPS scheduling discipline provides bandwidth guarantee, if a flow's arrival rate does not exceed this guaranteed service rate, all its traffic can traverse the port. Therefore, we introduce a new rule here whose priority is just below Rule 1'. The pre-condition to trigger the new rule is the existence of a flow that has a settled arrival rate no greater than its guaranteed service rate. When this condition holds, the settled arrival rate can be propagated through the port and then used to settle the corresponding flow variable at the next port on the flow's path. The new rule, called Rule 2', is illustrated in Table 5.15.

Rules 2, 3 and 4 in the original algorithm are relevant to a transparent port. Since GPS scheduling policy is work-conserving, if the aggregate arrival rate into a port is known not to exceed the link bandwidth associated with it, all the input traffic can be served. This suggests that all these rules are still effective on a port with the GPS scheduling policy.

Rule 5 in Table 5.5 establishes the upper bound on a flow variable at a port when the corresponding flow has a settled arrival rate at the upstream port that is in the *unresolved* state. In order to compute the upper bound, we assume that all flow variables not in the *settled* state at the upperstream port have arrival rate 0. Then, we apply the corresponding scheduling discipline to compute the departure rate for the flow under consideration. The derived departure rate must be no smaller than the real departure rate, because if flows whose arrival rates into the upstream port are not settled have positive rates, they will only squeeze bandwidth from flows whose arrival rates have already been settled. Therefore, the derived departure rate must be an upper bound on the real departure rate. The GPS scheduling discipline does not violate the above reasoning, which suggests that Rule 5 in Table 5.5 is still effective after modifying the way to compute the departure rate from Equation (5.3) to the GPS algorithm.

Similarly, we can modify Rule 6 in Table 5.6 to reflect the GPS service discipline. Recall that

this rule establishes the upper bound on a flow variable at a port when the corresponding flow variable at the upstream port is in the *bounded* state. The upper bound is computed by assuming that all flow variables in the *bounded* state but the one under analysis have rate 0. Their true rates may be positive, but this will only take away portion of bandwidth allocated to the flow under consideration, making its true departure rate smaller than the derived departure rate under the assumption. Therefore, when the GPS scheduling policy is considered, we can simply change the way to compute the departure rate in Rule 6 from Equation (5.3) to the GPS algorithm.

5.9 Related Work

There are a few fluid-oriented simulation models in which simulation time advances by constant time steps. The time-driven fluid-oriented model in [141] simulates the behavior of every queue in the network by forming the upper and lower bounds for both its backlog and its departure process. At every time step, the ingress process at a queue is modeled by two extreme cases: all traffic arrives immediately after the previous time step or all traffic arrives immediately before the current time step. Apparently, this differs from our approach in which ingress traffic of a flow is flattened between any two successive time steps. Their method establishes the upper and lower bounds on the backlog and the departure process with the upper and lower bounds on the arrival process at every queue. In a feed-forward network, the method assumes that the propagation delay of any link is 0; in a network with feedbacks where circular dependence might exist, non-zero propagation delay is assumed. The utility of the method is contingent on the closeness between the upper and lower bounds established on the backlogs or the departure processes: if the difference is sufficiently small throughout the simulation, then a good estimate on the sample path of the system is obtained. In addition, there is a tradeoff between the simulation efficiency and the time step size: a larger time step size reduces the frequency of the computation but may introduce larger gaps between the upper and lower bounds for some processes, and a small time step size improves the simulation accuracy but imposes more computation load.

In [85], a time-stepped fluid-oriented algorithm is proposed to simulate TCP traffic in large-scale IP networks. The foundation of this algorithm is a few sets of closely coupled nonlinear differential equations. The first set of equations characterizes how TCP adapts its congestion window size to the current network condition, including the round trip time and packet loss rate. Another set of equations captures the dynamics of the backlog in a queue as a function of its aggregate arrival rate. In their approach, AQM (Active Queue Management) policies such as RED (Random Early Dropping) are considered; hence, the dropping probability at a queue, as a function of the backlog, can also be described with differential equations. The differential equations that are applied on all the TCP flows and all the queues in the topology form a system of nonlinear differential equations. A time-stepped algorithm is proposed to derive the sample path of every component in the network throughout the simulation. At every time step, the Runge-Kutta algorithm [32] is applied to solve the system of differential equations numerically. Then, all state variables, including congestion window sizes, queue lengths, and loss probabilities at congestion links, are updated for the next time step. In order to reduce the complexity of the model, an optimization similar to transparent

port identification in our approach is applied: at every time step, uncongested links are identified and then removed from the network model, reducing the number of equations that need to be solved. A link is identified as uncongested if the sum of two components does not exceed its capacity: one component is the sum of the capacities of all upstream queues, and the other is the sum of the arrival rates of all the TCP flows originating from the starting endpoint of the link. Compared with our approach which establishes upper bounds on the flow variables on each flow's path, their method is less aggressive on reducing model complexity. As other time-stepped simulation techniques, their approach needs to find a tradeoff between accuracy and performance. A smaller time step provides more accurate solution, but needs more computation work. Moreover, owing to TCP's sensitivity to the round trip time, any step size larger than average round trip time inevitably affects the simulation accuracy. This differs from our approach which considers traffic behavior at coarse time scales.

An example of integrating time-stepped fluid-oriented traffic simulation into the same network simulator as packet-oriented traffic simulation is MAYA [142][144]. The analytical fluid-based TCP model in [95] and a packet-oriented simulator, QualNet[4], coexist in the same framework. Statistics of traffic flows represented as packet-oriented models are averaged over each time step, and they affect the network parameters associated with the fluid-based TCP model. On the other hand, the analytical fluid-based TCP model is periodically resolved with an ODE(Ordinary Differential Equation) solver to produce new network statistics like queue length and packet dropping probabilities. Hence, the packet-oriented traffic is affected by these results in the next time step. The model to integrate time-stepped fluid-oriented traffic and packet-oriented traffic in MAYA bears a lot of similarity to ours, although in our approach the foreground traffic involves not only packet-oriented traffic but event-driven fluid-oriented traffic.

Fixed point solutions are not new in the area of networking research. For instance, in [15], the fixed point approach is used to study the performance of a large number of TCP flows traversing through a network of AQM queues. In this approach, the throughput of a TCP flow is expressed as a function of network condition metrics, such as packet loss rate and queue lengths. For a typical AQM queueing policy like RED, the probability with which a packet is dropped or marked is a function of the current queue length. The objective is to find a fixed point solution to all the relevant equations, in which queue lengths are variables to be determined. When all the queue lengths at the fixed point are known, the throughput of each TCP flow can be derived. Although it was shown that the fixed point approach is able to approximate TCP behavior in AQM networks, its own efficiency is not particularly considered. This is different from our approach, in which performance is emphasized because of the motivated applications. In addition, fixed point method is a tool often used in estimating blocking probabilities in circuit switched networks [127]. In [82], fixed point approach is used to approximate the end-to-end blocking probability in a lossy multi-class network adopting a least loaded routing mechanism. The performance is evaluated with empirical experiments, because it is found difficult to theoretically establish the number of iterations needed before a fixed point is reached. This is similar to the challenge we have also encountered.

[4]http://www.scalable-networks.com

5.10 Summary

In this chapter, we have presented a time-stepped coarse-grained traffic simulation technique. The work is motivated by the real time constraint on simulating background traffic in large-scale network simulation. The sheer volume of background traffic in a large network excludes alternative methods such as packet-oriented traffic simulation. We propose an approach to computing background traffic load on each link periodically, assuming that traffic demands among ingress-egress pairs are updated at every time step. Solutions in diverse settings are analyzed, and they form the foundation of our algorithm.

The key idea of our algorithm is to establish upper bounds on the input flow variables at every port, hoping more ports can be identified as transparent. Port transparency enables us to settle some flow variables without resolving the ports. Only after this method fails to settle some flow variables, will fixed point iterations be applied to compute their rates. While we have not proved convergence of our algorithm, divergence has never been observed in experiments with realistic topologies,.

The algorithm is parallelized on a distributed-memory multiprocessor. Non-committal barrier synchronization is necessary for isolating different phases in the algorithm. It is observed that entering the barrier prematurely may lead to a lot of synchronization messages. We exploit some application-specific knowledge to postpone a processor's entrance into the barrier. From the experimental results with both fixed and scaled problem sizes, excellent scalability has been observed.

The algorithm is given in the settings of FIFO networks. However, with minor modifications, it can be used in other types of networks. We discussed how to modify the algorithm in a network which deploys the GPS scheduling discipline.

Chapter 6

Conclusions and Future Work

Research on issues in large-scale networks like the Internet demands high performance simulation tools. Real-time network simulation has been the foundation of many applications such as network emulation, online network control, and cyber-exercise. As simulation of large-scale networks at packet level imposes too much computation workload on the simulator, multi-resolution traffic modeling offers a viable solution. The main theme of this thesis is to develop an framework in which traffic represented at multiple abstraction levels can be efficiently simulated. In this chapter, we summarize the contributions made in this thesis and give some limitations. Finally, we envision the potential research directions along which the current work can be extended.

6.1 Contributions

In this dissertation, we have presented a multi-resolution traffic simulation framework that integrates traffic models represented at three abstraction levels: packet-oriented models, event-driven fluid-oriented traffic models, and time-stepped fluid-oriented traffic models. The major contributions made in this dissertation is to address some important problems regarding the efficiency, accuracy, and scalability of this framework.

A well known efficiency problem with event-driven fluid-oriented traffic simulation is the "ripple effect". This phenomenon can cause explosion of fluid rate changes when multiple ports are congested in the network. We have proposed a rate smoothing technique to mitigate this phenomenon. This approach exploits the insensitive period which fluid rate changes have to suffer when they traverse on a link. In order to prevent overly high errors resulting from this method, we further bring forward a constrained version to balance accuracy and efficiency.

Integrating mixed traffic representations at the same port is a delicate process. It involves how to compute the backlog in the buffer, how to allocate bandwidth to packet-oriented and fluid-oriented traffic, and how to determine the packet loss rate when the buffer overflows. Through a lot of experimentation, we have implemented a hybrid port model that is able to produce accurate results under varied conditions. Speedups at orders of magnitude over pure packet-level simulation have

been observed from the simulation results.

Given the dominance of TCP traffic in the Internet, it is a consequential objective to improve the efficiency of its simulation. We have extended an existing fluid-based TCP model in our framework. Simulation results tell us that under light or medium traffic load, the hybrid simulator provides significant speedups over the pure packet-level simulator without impairing the accuracy. Under extremely heavy traffic load, however, the hybrid simulator may be outperformed.

Real-time simulation of the huge amount of traffic in a large-scale network is a highly challenging undertaking. We have developed a time-stepped simulation technique that simulates large-scale network traffic at coarse time scales. This approach assumes that traffic reaches convergence within an ignorable time interval compared with the large time step. It addresses how to efficiently compute the traffic load on each link when the traffic demands at the edge of the network change. Our approach relies on fixed point iterations when flow variables form circular dependencies, but aggressively reduces their number before the iterative solution is applied. A further optimization is to merge flows that are destined to the same place in the computation. The technique renders it possible to simulate a topology with 3 ISP backbones in real time on an ordinary PC.

We have developed a parallel algorithm to simulate coarse-grained network traffic in a large-scale network. Non-committal barrier synchronization has shown to be necessary in the algorithm. In order to reduce the synchronization cost, we have exploited application-specific knowledge to postpone a processor's entrance into the barrier. Simulation results with both fixed and scaled problem sizes illustrate that our parallel algorithm can be excellently scaled up to a fair number of processors.

6.2 Limitations

Throughout this dissertation, we have seen that simulation techniques developed in the multi-resolution traffic simulation framework have achieved performance improvement over traditional packet-oriented traffic simulation at varied orders of magnitude. However, we are also aware of the following limitations of the solutions proposed in this thesis work.

1. Some models proposed in this thesis have one or more input parameters. In order to achieve efficient simulation with these models, we need configuration of the input parameters properly. For instance, the constrained rate smoothing technique discussed in Section 4.2.3 requires to configure the constraint on the time interval within which fluid rate changes are allowed to be smoothed. If this constraint is too large, the simulation may be equivalent to the unconstrained rate smoothing technique and thus produce results with high errors. If this constraint is too small, fluid rate changes are seldom smoothed, providing little help on improving the simulation efficiency. Proper configuration on such parameters, though, needs a full understanding of both the topology and the end application's requirements.

2. As many experimental results in this thesis suggest, there often exist a tradeoff between simulation efficiency and simulation accuracy. Sometimes, it is difficult, however, to establish the

bounds on the errors resulting from an approximation or simplification. Intensive experiments are necessary to examine its impact under diverse settings, which itself can be computationally expensive.

3. The discrete event fluid port discussed in Section 4.2.1 is limited to the FIFO queueing policy. It can also be extended for other queueing disciplines like GPS. However, we have also realized that it is difficult to model a fluid port implementing some AQM queueing policies like RED in our framework. Under these policies, the packet loss rate is nondeterministic function of the current backlog in the buffer. Hence, the departure rate of a flow is no longer a piece-wise constant rate function, making it difficult to apply discrete event models.

4. The time-stepped coarse-grained traffic simulation technique described in Section 5.5 lacks a theoretical foundation on its convergence behavior. Although we have never observed divergence behavior of the algorithm in experiments, we are unable to provide a rigorous proof on its convergence property in this thesis. Currently we set a heuristic-based upper bound on the number of itreations allowed in the fixed point computation.

5. We lack an effective and efficient topology partitioner for the parallel algorithm that simulates large-scale network traffic at coarse time scales in a time-stepped fashion. It is imaginable that the algorithm, when used to simulate a realistic topology with realistic traffic patterns, may suffer significant performance degradation, if the computation workload over the participating processors are not balanced.

These limitations suggest that the techniques proposed in this thesis have not provided an end solution to the motivating problems mentioned in the opening chapter. Some related problems still need to be addressed. However, as the results in the thesis tell us, multi-resolution traffic modeling offers hope that a realistic large-scale network like the Internet can be efficiently simulated with fidelity.

6.3 Future Work

One direction for future research is to address the limitations mentioned in Section 6.2. Besides that, we identify several directions along which the work in this thesis can be extended.

6.3.1 Simulation-Based Online Network Control

A lot of simulation results presented in this thesis show that a large-scale network can be simulated faster than real time. This suggests a promising field in which faster-than-real-time network simulation can be exploited to control the network operation in an online manner. For instance, in a network that supports multi-path routing, how to split traffic among alternative paths is a decision that depends on the current traffic demands and the operational objective of the network. One objective commonly used is to balance the traffic load among the links in the network so that packet losses are minimized. A faster-than-real-time network simulator can be used in the optimization

process to evaluate alternative routing plans. Compared with some other online network control methods in which the optimization process is done on real networks, the simulation-based approach isolates the optimization process from real networks, thus minimizing its effect on ongoing network services.

6.3.2 Load Balancing in High-Fidelity Network Simulation

Modeling network traffic at high abstraction level usually provides better simulation performance, but in many cases, such execution speedup comes at a cost on compromising the simulation accuracy or losing details at a certain level. When high-fidelity simulation is necessary, parallel simulation may remain as the only viable solution to its scalability. In order to optimize the efficiency of the parallel network simulation, balancing the workload over all the processors is of crucial importance. Traffic simulation at high abstraction level can be utilized to profile the traffic load throughout the simulation within a relatively short time, and such a traffic profile is helpful in improving workload balancing in the high-fidelity network simulation.

6.3.3 Large-Scale Distributed Application Traffic Simulation

In this thesis, we have developed a multi-resolution traffic simulation framework. This framework provides a platform for simulating not only the interaction among the traffic injected from the distributed applications at the edge of the network but also the interaction between the distributed application traffic and other types of traffic existing in the network. There are many high-end applications that are widely spread in the Internet and have contributed or will contribute to a significant portion of the Internet traffic, such as peer-to-peer file sharing and Grid applications. Simulation has been an important tool in gaining deep insight into their behavior, but the network model used in the simulation is often overly simplified because the high fidelity simulation requires long execution time. One research direction is to model the behavior of these distributed applications with fluid-oriented models and simulate their traffic in our multi-resolution traffic simulation framework. Currently we are working on large-scale worm traffic simulation. It is a perfect example of such distributed applications.

6.3.4 Multi-Resolution Wireless Network Simulation

Throughout this thesis, we limit our discussion to traffic simulation in wireline networks like the Internet. One research direction that can be pursued is to simulate wireless network traffic with multi-resolution models. Some analytical fluid-based models have been developed in [66] for wireless network simulation. It remains further investigation whether event-driven fluid-oriented models are a viable approach to improving its performance.

6.4 Final Remarks

In this dissertation we have presented a multi-resolution traffic simulation framework in which network traffic represented at different abstraction levels can be simulated. We have addressed some important issues in this framework, including its efficiency, accuracy and scalability. Our work has provided a powerful simulation tool for researchers to investigate Internet-related problems.

Bibliography

[1] J. S. Ahn and P. B. Danzig. Packet network simulation: Speedup and accuracy versus timing granularity. *IEEE/ACM Transactions on Networking*, 4(5):743–757, October 1996.

[2] E. Al-Shaer and H. Hamed. Discovery of policy anomalies in distributed firewalls. In *Proceedings of 23rd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'04)*, Hong Kong, China, March 2004.

[3] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: An experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, November 2002.

[4] D. Awduche, A. Chiu, A. Elwalid, I. Widjaja, and X. Xiao. Overview and principles of Internet traffic engineering. RFC 3272, May 2002.

[5] R. Bagrodia, R. Meyer, M. Takai, Y. Chen, X. Zeng, J. Martin, B. Park, and H. Song. Parsec: A parallel simulation environment for complex systems. *IEEE Computer*, 31:77–85, October 1998.

[6] J. Banks, II J. S. Carson, B. L. Nelson, and D. M. Nicol. *Discrete-event system simulation*. Prentice Hall, December 2004.

[7] S. M. Bellovin. Distributed firewalls. *USENIX ;login:*, pages 39–47, November 1999.

[8] J. Beran. Statistical methods for data with long-range dependence. *Statistical Science*, 7(4):404–427, 1992.

[9] F. Berman. High-performance schedulers. In I. Foster and C. Kesselman, editors, *The Grid: blueprint for a new computing infrastructure*, pages 279–309. Morgan-Kaufmann, 1999.

[10] G. E. P. Box and G. M. Jenkins. *Time Series Analysis: Forecasting and Control*. Prentice Hall, 1976.

[11] J. V. Briner. *Parallel mixed-level simulation of digital circuits using virtual time*. PhD thesis, Duke University, Durham, NC, USA, 1990.

[12] R. Brown. Calendar queues: A fast o(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, October 1988.

[13] R. E. Bryant. Simulation of packet communication architecture computer systems. Technical Report MIT-LCS-TR-188, Massachusetts Institute of Technology, 1977.

[14] T. Bu, N. Duffield, F. L. Presti, and D. Towsley. Network tomography on general topologies. *ACM SIGMETRICS Performance Evaluation Review*, 30(1):21–30, June 2002.

[15] T. Bu and D. Towsley. Fixed point approximations for tcp behavior in an aqm network. *ACM SIGMETRICS Performance Evaluation Review*, 29(1):216–225, June 2001.

[16] R. Callon. Use of osi is-is for routing in tcp/ip and dual environments. Request For Commends (Standard) RFC 1195, Internet Engineering Task Force, December 1990.

[17] J. Cao, D. Davis, S. Wiel, and B. Yu. Time-varying network tomography: Router link data. *Journal of the American Statistical Association*, 95(452):1063–1075, February 2000.

[18] R. Castro, M. Coates, G. Liang, R. Nowak, and . Yu. Network tomography: Recent developments. *Statistical Science*, 19(3):499–517, August 2004.

[19] D. Caughlin. A metamodeling approach to model abstraction. In *Proceedings of the Furth Dual Use and Technologies Conference*, pages 387–396, San Francisco, California, 1994.

[20] J. Chandrashekar, Z. Duan, Z. Zhang, and J. Krasky. Limiting path exploration in BGP. In *Proceedings of 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'05)*, Miami, FL, USA, 2005.

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

[22] S. Chen and K. Nahrstedt. An overview of quality-of-service routing for the next generation high-speed networks: Problems and solutions. *IEEE Network, Special Issue on Transmission and Distribution of Digital Video*, Nov./Dec. 1998.

[23] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Petterson, M. Wawrzoniak, and M. Bowman. Planetlab: an overlay testbed for broad-coverage services. *ACM Computer Communications Review*, 33(3), July 2003.

[24] L. Chwif and R. J. Paul. On simulation model complexity. In *Proceedings of the 2000 Winter Simulation Conference*, Orlando, FL, USA, December 2000.

[25] K. Claffy, G. Miller, and K. Thompson. The nature of the beast: Recent traffic measurements from an Internet backbone. In *Proceedings of INET'98*, 1998.

[26] M. J. Coates and R. Nowak. Network tomography for internal delay estimation. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, Salt Lake City, Utah, USA, May 2001.

[27] M. J. Coates and R. Nowak. Sequential Monte Carlo inference of internal delays in nonstationary communication networks. *IEEE Transactions on Signal Processing, Special Issue on Monte Carlo Methods for Statistical Singal Processing*, pages 366–376, March 2002.

[28] D. E. Comer. *Internetworking with TCP/IP*, volume I of *Principles, protocols, and Architecture*. Prentice Hall, Inc., third edition, 1995.

[29] J. C. Comfort. The simulation of a microprocessor based event set processor. In *Proceedings of the 14th annual symposium on Simulation*, Tampa, Florida, USA, 1981.

[30] A. Cranas and J. Dugundji. *Fixed Point Theory*. Springer-Verlag, New York, USA, 2003.

[31] D. J. Daley and J. Gani. *Epidemic Modeling: An Introduction*. Cambridge University Press, Cambridge, UK, 1999.

[32] J. W. Daniel and R. E. Moore. *Computation and theory in ordinary differential equations*. W. H. Freeman, San Francisco, 1970.

[33] N. Feamster, J. Borkenhagen, and J. Rexford. Guidelines for interdomain traffic engineering. *ACM SIGCOMM Computer Communications Review*, 33(5):19–30, October 2003.

[34] S. L. Ferenci, R. M. Fujimoto, M. H. Ammar, K. Perumalla, and G. F. Riley. Updateable simulation of communication networks. In *Proceedings of the sixteenth workshop on Parallel and distributed simulation(PADS'02)*, Washington, D.C., USA, 2002.

[35] A. Ferscha and S. K. Tripathi. Parallel and distributed simulation of discrete event systems. Technical Report CS-TR-3336, University of Maryland, MD, USA, August 1994.

[36] M. Fomenkov, K. Keys, D. Moore, and K Claffy. Longitudinal study of Internet traffic in 1998-2003. In *Proceedings of the winter international synposium on Information and communication technologies*, 2004.

[37] Ian Foster. *Desigining and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[38] F. K. Frantz. A taxonomy of model abstraction techniques. In *Proceedings of the 1995 Winter simulation Conference*, Arlington, VA, USA, 1995.

[39] V. S. Frost and B. Melamed. Traffic modeling for telecommunicaitons networks. *IEEE Communication Magazine*, 3, March 1994.

[40] R. M. Fujimoto. Time warp on a shared memory multiprocessor. *Transactions of the Society for Computer Simulation*, 6:211–239, July 1989.

[41] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(20):30–53, October 1990.

[42] L. Gao and J. Rexford. Stable internet routing without global coordination. *IEEE/ACM Transactions on Networking*, 9(6):681–692, December 2001.

[43] L. Garber. Denial-of-service attacks rip the Internet. *IEEE Computer*, 33(4):12–17, April 2000.

[44] P. Glasserman, P. Heidelberger, P. Shahabuddin, and T. Zajic. Splitting for rare event simulation: analysis of simple cases. In *Proceedings of the* 28th *conference on Winter simulation*, Coronado, CA, USA, 1996.

[45] NSA Security Recommendation Guides. Defense in depth: a practical strategy for achieving information assurance in today's highly networked environments, 2003.

[46] Y. Guo, W. Gong, and D. Towsley. Time-stepped hybrid simulation (TSHS) for large scale networks. In *Proceedings of 19th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'00)*, Tel Aviv, Israel, March 2000.

[47] C. Hedrick. Routing information protocol. Request For Commends (Standard) RFC 1058, Internet Engineering Task Force, June 1988.

[48] J. O. Henriksen. An improved events list algorithm. In *Proceedings of the 1977 Winter Simulation conference*, pages 547–557, 1977.

[49] P. Huang and J. Heidemann. Minimizing routing state for light-weight network simulation. In *Proceeding of the* 9th *International Symposium on Modeling, Analysis and Simulation on Computer and Telecommunication Systems (MASCOTS'01)*, Cincinnati, Ohio, USA, August 2001.

[50] G. Huston. Analyzing the Internet's bgp routing table. *The Internet Journal Protocol*, 4(1):2–15, 2001.

[51] M. Hybinette and R. M. Fujimoto. Cloning parallel simulations. *ACM Transactions on Modeling and Computer Simulation*, 11(4):378–407, October 2001.

[52] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith. Implementing a distributed firewall. In *Proceedings of 7th ACM Conference on Computer and Communications Security(CCS'00)*, November 2000.

[53] A. T. Ogielski J. Cowie, B. J. Premore, and Y. Yuan. Internet worms and global routing instabilities. In *Proceedings of SPIE*, volume 4868, July/August 2002.

[54] V. Jacobson. Congestion avoidance and control. In *Proceedings of the 1988 conference on Applications, technologies, architectures, and protocols for computer communication(SIGCOMM'88)*, Stanford, CA, USA, August 1988.

[55] V. Jacobson. Berkeley tcp evolution from 4.3-tahoe to 4.3-reno. In *Proceedings of the Eighteenth Internet Engineering Task Force*. 1990.

[56] V. Jacobson. Modified tcp congestion avoidance algorithm. ftp://ftp.isi.edu/end2end/end2end-interest-1990.mail, April 30 1990.

[57] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*, chapter Part VI: Queueing Models. Wiley-Interscience, New York, NY, USA, April 1991.

[58] D. R. Jefferson, B. Beckman, F. Weiland, L. Blume, M. Dilorento, P. Hontalas, P. Reiher, K. Sturdevant, J. Tupman, J. Wedel, and H. Younger. The time warp operating system. In *Proceedings of the 11th Symposium on Operating Systems Principles*, volume 21, pages 77–93, November 1987.

[59] D. R. Jefferson and H. Sowizral. Fast concurrent simulation using the time warp mechanism, part i: Local control. Technical Report N-1906-AF, RAND Corporation, December 1982.

[60] D. Katabi and C. Blake. A note on the stability requirements of adaptive virtual queue. Technical Report MIT-LCS-TM-626, Laboratory for Computer Science, Massachusetts Institute of Technology, 2002.

[61] G. Kesidis, A. Singh, D. Cheung, and W. Kwok. Feasibility of fluid-event-driven simulation for ATM networks. In *Proceedings of IEEE Globecom'96*, London, GB, November 1996.

[62] K. Khanin, D. Khmelev, A. Rybko, and A. Vladimirov. Steady solutions for FIFO networks. *Moscow Mathematical Journal*, 1(3):407–419, September 2001.

[63] C. Kiddle. *Scalable Network Emulation*. PhD thesis, Department of Computer Science, University of Calgary, 2004.

[64] C. Kiddle, R. Simmonds, C. Williamson, and B. Unger. Hybrid packet/fluid flow network simulation. In *Proceedings of the 19th Workshop on Parallel and Distributed Simulation (PADS'03*, San Diego, CA, USA, June 2003.

[65] T. Kiesling and J. Luethi. Towards time-parallel road traffic simulation. In *Proceedings of the 19th Workshop on Parallel and Distributed Simulation (PADS'05)*, Monterey, CA, USA, June 2005.

[66] H. Kim and J. C. Hou. In *Proceedings of the joint international conference on Measurement and modeling of computer systems(SIGMETRICS)*, Simulation tools and analysis techniques, New York, NY, USA, 2004.

[67] L. Kleinrock. *Queueing Systems Volume 1: Theory*. John Wiley, Inc., New York, NY, USA, 1975.

[68] K. Kumaran and D. Mitra. Performance and fluid simulations of a novel shared buffer management system. *ACM Transactions on Modeling and Computer Simulation*, 11(1):43–75, January 2001.

[69] C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian. Delayed Internet routing convergence. *IEEE/ACM Transactions on networking*, 9(3):293–306, June 2001.

[70] K. Lai and M. Baker. Measuring link bandwidths using a deterministic model of packet delay. In *Proceedings of the 2000 conference on Applications, technologies, architectures, and protocols for computer communication(SIGCOMM'00)*, Stockholm, Sweden, August 2000.

[71] A. M. Law. How to conduct a successful simulation study. In *Proceedings of the 2003 Winter Simulation Conference*, New Orleans, LA, USA, December 2003.

[72] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Upper Saddle River, NJ, USA, 1984.

[73] J. Y. Lee, S. Kim, and D. K. Sung. Fluid simulation for self-similar traffic in generalized processor sharing servers. *IEEE Communications Letters*, 7(6), June 2003.

[74] W. E. Leland and D. V. Wilson. High time-resolution measurement and analysis of lan traffic: Implication for lan interconnection. In *Proceedings of 10th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'91)*, Florida, USA, April 1991.

[75] M. Liljenstam, J. Liu, D. M. Nicol, Y. Yuan, G. Yan, and C. Grier. Rinse: the real-time immersive network simulation environment for network security exercises. In *Proceedings of the 19th Workshop on Parallel and Distributed Simulation (PADS'05)*, Monterey, CA, USA, June 2005.

[76] M. Liljenstam and D. M. Nicol. On-demand computation of policy based routes for large-scale network simulation. In *Proceedings of the 2004 Winter Simulation Conference*, Washington, D.C., USA, December 2004.

[77] M. Liljenstam, D. M. Nicol, V. Berk, and R. Gray. Simulating realistic network worm traffic for worm warning system design and testing. In *Proceedings of the 2003 ACM Workshop on Rapid Malcode (WORM 2003)*, Washington DC, USA, October 2003.

[78] M. Liljenstam and D. M. Nicol Y. Yuan, B. J. Premore. A mixed abstraction level simulation model of large-scale internet worm infestations. In *Proceedings of the Tenth IEEE/ACM Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'02)*, Fort Worth, TX, USA, October 2002.

[79] B. Liu, D. R. Figueiredo, Y. Guo, J. Kurose, and D. Towsley. A study of networks simulation efficiency: Fluid simulation vs. packet-level simulation. In *Proceedings of 20th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'01)*, Alaska, USA, April 2001.

[80] B. Liu, Y. Guo, J. Kurose, D. Towsley, and W. Gong. Fluid simulation of large scale networks: Issues and tradeoffs. In *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications*, volume IV, pages 2136–2142, 1999.

[81] J. Liu and D. M. Nicol. Lookahead revisited in wireless network simulations. In *Proceedings of the 16th Workshop on Parallel and Distributed Simulation (PADS'02)*, pages 79–88, Washington, D.C., USA, May 2002.

[82] M. Liu and J. S. Baras. Fixed point approximation for multirate multihop loss networks with state-dependent routing. *IEEE/ACM Transactions on Networking*, 12(2):361–374, April 2004.

[83] X. Liu and A. A. Chien. Traffic-based load balance for scalable network emulation. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, Phoenix, Arizona, USA, November 2003.

[84] X. Liu and A. A. Chien. Realistic large-scale online network simulation. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, Pittsburgh , Pennsylvania, USA, November 2004.

[85] Y. Liu, F. L. Presti, V. Misra, D. Towsley, and Y. Gu. Scalable fluid models and simulations for large-scale ip networks. *ACM Transactions on Modeling and Computer Simulation*, 14(3):305–324, July 2004.

[86] Y-B. Liu and E. D. Lazowska. Reducing the state saving overhead for time warp parallel simulation. Technical Report 90-02-03, Dept. of Computer Science, University of Washington, Seattle, Washington, USA, February 1990.

[87] S. H. Low, F. Paganini, J. Wang, S. Adlakha, and J. C. Doyle. Dynamics of TCP/AQM and a scalable control. In *Proceedings of 21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'02)*, New York, NY, USA, June 2002.

[88] B. D. Lubachevsky. Scalability of the bounded lag distributed discrete event simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, volume 21, pages 100–107, March 1989.

[89] M. T. Lucas, B. J. Dempsey, D. E. Wrege, and A. C. Weaver. An efficient background traffic model for wide area network simulation. In *Proceedings of IEEE Global Telecommunications Conference*, Phoenix, Arizona, USA, November 1997.

[90] B. Maglaris, D. Anastassiou, P. Sen, G. Karlsson, and J. D. Robbins. Performance models of statistical multiplexing in packet video communications. *IEEE Transactions on Communications*, 36(7), 1988.

[91] Z. M. Mao, R. Govindan, G. Varghese, and R. H. Katz. Route flap damping exacerbates Internet routing convergence. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communication(SIGCOMM'02)*, Pittsburgh, Pennsylvania, USA, August 2002.

[92] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The macroscopic behaviour of the tcp congestion avoidance algorithm. *ACM Computer Communication Review*, 27(3), July 1997.

[93] B. Melamed, S. Pan, and Y. Wardi. Hybrid discrete-continuous fluid-flow simulation. In *Proceedings of the SPIE International Symposium on Information Technologies and Communications (IT-COM 01)*, Scalability and Traffic Control in IP Networks, Denver, Colorado, USA, August 2001.

[94] J. Misra. Distributed discrete-event simulation. *ACM Computing Surveys*, 18(1):39–65, March 1986.

[95] V. Misra, W. Gong, and D. Towsley. A fluid-based analysis of a network of AQM routers supporting TCP flows with an application to RED. In *Proceedings of the 2000 conference on Applications, technologies, architectures, and protocols for computer communication(SIGCOMM'00)*, Stockholm, Sweden, September 2000.

[96] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the slammer worm. *IEEE Security and Provacy Magazine*, 1(4):33–39, July 2003.

[97] D. Moore, C. Shannon, and J. Brown. Code-red: a case study on the spread and victims of an Internet worm. In *Proceedings of ACM/UNENIX Internet Measurement Workshop*, France, November 2002.

[98] D. Moore, C. Shannon, and K. Claffy. Code-red: A case study on the spread and victims of an Internet worm. In *Proceedings of the Internet Measurement Workshop(IMW'02)*, Marseille, France, November 2002.

[99] C. J. P. Moschovitis, H. Poole, T. Schurler, and T. M. Senft. *History of the Internet: A Chronology, 1843 to the Present*. ABC-Clio, 1999.

[100] J. Moy. OSPF version 2. Request For Commends (Standard) RFC 2328, Internet Engineering Task Force, April 1998.

[101] D. M. Nicol. Parallel discrete-event simulation of fcfs stochastic queueing networks. *ACM SIGPLAN Notice*, 23(9):124–137, September 1988.

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

[103] D. M. Nicol. Non-committal barrier synchronization. *Parallel Computing*, 21:529–549, 1995.

[104] D. M. Nicol. Discrete event fluid modeling of tcp. In *Proceedings of the 2001 Winter Simulation Conference*, Arlington, VA, USA, December 2001.

[105] D. M. Nicol and R. Fujimoto. Parallel simulation today. *Annals of Operations Research*, 53:249–286, 1994.

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

[107] D. M. Nicol, S. W. Smith, and M. Zhao. Evaluation of efficient security for bgp route announcements using parallel simulation. *Elsevier Simulation Modelling Practice and Theory Journal, special issue on Modeling and Simulation of Distributed Systems and Networks*, 12(3-4):187–216, July 2004.

[108] D. M. Nicol and G. Yan. Discrete event fluid modeling of background tcp traffic. *ACM Transactions on Modeling and Computer Simulation*, 14(3):1–39, July 2004.

[109] D. M. Nicol and G. Yan. Simulation of network traffic at coarse timescales. In *Proceedings of the 19th Workshop on Parallel and Distributed Simulation (PADS'05)*, Monterey, CA, USA, June 2005.

[110] A. M. Odlyzko. Internet traffic growth: sources and implications. In B. B. Dingel, W. Weiershausen, A. K. Dutta, and K.-I. Sato, editors, *Optical Transmission Systems and Equipment for WDM Networking II*, volume 5247 of *Proceedings of SPIE*, 2003.

[111] P. S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers, Inc., 1997.

[112] E. H. Page, D. M. Nicol, O. Balci, R. M. Fujimoto, P. A. Fishwick, P. L'Ecuyer, and R. Smith. Panel: Strategic directions in simulation research. In *Proceedings of the 1999 Winter Simulation Conference*, Phoenix, AZ, USA, December 1999.

[113] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, June.

[114] K. Park. Future directions and open problems in performance evaluation and control of self-similar network traffic. In *Self-Similar Network Traffic and Performance Evaluation*. Wiley-Interscience, July 2000.

[115] D. Pei, X. Zhao, L. Wang, D. Massey, A. Mankin, S. F. Wu, and L. Zhang. Improving bgp convergence through consistency assertions. In *Proceedings of 21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'02)*, New York, NY, USA, June 2002.

[116] K. Perumalla, R. M. Fujimoto, and A. T. Ogielski. Ted - a language for modeling telecommunication networks. *ACM SIGMETRICS Performance Evaluation Review*, 25(4):4–11, March 1998.

[117] K. Perumalla and S. Sundaragopalan. High-fidelity modeling of computer network worms. In *Proceedings of 20th Annual Computer Security Applications Conference*, Tucson, Arizona, USA, December 2004.

[118] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[119] R. Radhakrishnan, D. E. Martin, M. Chetlur, D. M. Rao, and P. A. Wilsey. An object-oriented time warp simulation kernel. In *Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments*, December 1998.

[120] D. M. Rao and P. A. Wilsey. Simulation of ultra-large communication networks. In *Proceedings of the Seventh IEEE/ACM Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'99)*, College Park, Maryland, USA, October 1999.

[121] R. Rönngren and R. Ayani. A comparative study of parallel and sequential priority queue algorithms. *ACM Transactions on Modeling and Computer Simulation*, 7(2):157–209, April 1997.

[122] R. Rönngren, J. Riboe, and R. Ayani. Lazy queue: new approach to implementing the pending event set. *International Journal of Computer Simulation*, 3:303–332, 1993.

[123] Y. Rekhter and T. Li. A border gateway protocol 4(bgp-4). RFC 1771, Internet Engineering Task Force, March 1995.

[124] G. F. Riley and M. H. Ammar. Simulating large networks - how big is big enough? In *Proceedings of First International Conference on Grand Challenges for Modeling and Simulation*, January 2002.

[125] G. F. Riley, M. H. Ammar, and R. M. Fujimoto. In *Proceeding of the* 8th *International Symposium on Modeling, Analysis and Simulation on Computer and Telecommunication Systems (MASCOTS'00)*, San Francisco, CA, USA, September 2000.

[126] G. F. Riley, T. M. Jaafar, and R. M. Fujimoto. Integrated fluid and packet network simulations. In *Proceedings of the 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'02)*, Fort Worth, TX, USA, October 2002.

[127] K. W. Ross. *Multiservice loss networks for boradband telecommunication networks*. Springer Verlag, 1995.

[128] P. Sen, B. Maglaris, N.-E. Rikli, and D. Anastassiou. Models for packet switching of variable-bit-rate video sources. *IEEE Journal on Selected Areas in Communications*, 7(5):865–869, 1989.

[129] R. E. Shannon. *Systems Simulation: the art and science*. Prentice-Hall, 1975.

[130] R. Simmonds, R. Bradford, and B. Unger. Applying parallel discrete event simulation to network emulation. In *Proceedings of the 14th Workshop on Parallel and Distributed Simulation (PADS'00)*, Bologna, Italy, 2000.

[131] D. D. Sleator and R. E. Tarjan. Self-adjusting heaps. *SIAM Journal on Computing*, 15(1):52–69, February 1986.

[132] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of ACM*, 32(3):652–686, July 1995.

[133] S. Staniford, V. Paxson, and N. Weaver. How to 0wn the internet in your spare time. In *Proceedings of the 11th USENIX Security Symposium (Security '02)*, August 2002.

[134] J. S. Steinman. Speedes: A unified approach to parallel simulation. In *Proceedings of the Sixth Workshop on Parallel and Distributed Simulation*, pages 75–84, Los Alamitos, CA, USA, 1992.

[135] I. Stoica, S. Shenker, and H. Zhang. *Core*-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks. In *Proceedings of the 1998 conference on Applications, technologies, architectures, and protocols for computer communication(SIGCOMM'98)*, Vancouver, B.C., Canada, September 1998.

[136] Y. Tang, K. S. Perumalla, R. M. Fujimoto, H. Karimabadi, J. Driscoll, and Y. Omelchenko. Optimistic parallel discrete event simulations of physical systems using reverse computation. In *Proceedings of the 19th Workshop on Parallel and Distributed Simulation (PADS'05)*, Montery, CA, USA, June 2005.

[137] B. W. Unger and G. A. Lomow. The telecom framework: A simulation environment for telecommunications. In *Proceedings of the 1993 Winter Simulation Conference*, December 1993.

[138] Y. Vardi. Network tomography: Estimating source-destination traffic intensities from link data. *Journal of the American Statistical Association*, 91(433):365–377, March 1996.

[139] K. Walsh and E. G. Sirer. Staged simulation: A general technique for improving simulation scale and performance. *ACM Transactions on Modeling and Computer Simulation*, 14(2):170–195, April 2004.

[140] Z. Wang and J. Crowcroft. Quality of service routing for supporting multimedia applications. *IEEE Journal of Selected Areas in Communications*, 14(7):1228–1234, September 1996.

[141] A. Yan and W. Gong. Time-driven fluid simulation for high-speed networks. *IEEE Transactions on Information Theory*, 45(5):1588–1599, July 1999.

[142] T. K. Yung, J. Martin, M. Takai, and R. Bagrodia. Integration of fluid-based analytical model with packet-level simulation for analysis of computer networks. In Robert D. van der Mei and Frank Huebner-Szabo de Bucs, editors, *Proceedings of SPIE*, volume 4523 of *Internet Performance and Control of Network Systems II*, pages 130–143, 2001.

[143] X. Zeng, R. Bagrodia, and M. Gerla. Glomosim: a library for parallel simulation of large-scale wireless networks. *ACM SIGSIM Simulation Digest*, 28(1):154–161, July 1998.

[144] J. Zhou, Z. Ji, M. Takai, and R. Bagrodia. Maya: Integrating hybrid network modeling to the physical world. *ACM Transactions on Modeling and Computer Simulation*, 14(2):149–169, April 2004.

[145] C. C. Zou, W. Gong, and D. Towsley. Code Red worm propagation modeling and analysis. In *Proceedings of the 9th ACM conference on Computer and communications security*, Washington DC, USA, November 2002.