在事件層平行方法下 NS2 網路模擬器的效能量測
The Performance of the NS2 Network Simulator using the Event-level Parallelism Approach

研 究 生：曾彥勳　　Student：Yan-Shiun Tzeng

指導教授：王協源　　Advisor：Shie-Yuan Wang

國 立 交 通 大 學

網 路 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Network Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

June 2007

Hsinchu, Taiwan, Republic of China

中華民國九十六年六月

在事件層平行方法下NS2網路模擬器的效能量測
The Performance of the NS2 Network Simulator using the Event-level Parallelism Approach

研究生：曾彥勳　　指導教授：王協源教授

國立交通大學
網路工程研究所

摘要

近年來，提升處理器的工作速度變的越來越難以有效改善程式的執行效能。主要的處理器製造商，如Intel、AMD等公司都進而轉往多核心處理器平台來增加額外的執行效能。因此具有多核心處理器平台的桌上型個人電腦或是筆記型電腦將會越來越普及，價錢也越來越被消費者所接受。

在這篇論文中，我們提出了新穎的平行模擬方法爲了加速網路模擬器在多核心架構上的模擬速度。論文中，我們將此方法實作應用在NS2網路模擬器上並做適當的設計調整。同時，我們量測NS2網路模擬器在各種不同的模擬條件之下效能提升的成果，並且討論系統未來的擴充性與發展改善的方向。

關鍵字：中央處理器、雙核心、四核心、多核心、網路模擬器、模擬、事件層平行模擬、執行緒、NS2、ELP。

The Performance of the NS2 Network Simulator using the Event-level Parallelism Approach

Student: Yan-Shiun Tzeng Advisors: Prof. Shie-Yuan Wang

Institute of Network Engineering
National Chiao-Tung University

ABSTRACT

In recent years increasing, CPU clock speed is becoming more and more difficult to effectively improve the performance. Major CPU venders such as Intel, AMD, etc. have all turned to multicore CPUs as the best way to gain additional performance. A desktop PC or NB which are with modern multicore systems will become more and more popular and its cost will become more and more inexpensive.

This thesis presents to a novel and general parallel simulation approach to increasing network simulation speeds on modern multicore systems. In this thesis, we present the design and implementation of this approach and apply it to the NS2 network simulator. We show the achieved performance speedups of the NS2 network simulator under various network conditions. Finally, several possible future developments are proposed.

Keywords: CPU, dual-core, quad-core, multicore, network simulator, simulation, NS2, ELP, event-level parallelism, Thread.

誌謝辭

在這兩年的研究所生涯，要感謝我的指導教授王協源老師與兩位博士班學長–周智良與林志哲，在老師與學長們的指導與經驗的傳授中，讓我學習到許多寶貴的實務經驗與研究方法，同時學業與專業技能也有很大幅度的成長，這些學習的過程對於未來職場生涯也將是無形中的資產與助力。同時也要感謝實驗室同儕，大家同甘共苦共體時艱，在論文寫作階段給予我許多寶貴的建議，實驗室的學弟也在我最繁忙的時刻協助我處理實驗室的事務，減輕我的負擔。

感謝系上計算機中心所有同學與學長，在我任職的這兩年內給予我不少的指導，尤其是蔡宗易學長、翁綜禧學長與朱漢威學長更不吝的教導我在計中任職所需要的各項技能與經驗，不僅在這段時間熟悉到更多系統管理方面的專業技術，更讓我學習到更階機房運作的相關知識。

最後，也要感謝我的母親，在我求學階段將行動不便的父親照顧的無微不至，讓我可以在無後顧之憂的情況專注研究所的課業，父親更是我的心靈導師與最佳模範，給予我許多人生的經驗與建議並教導我樂觀與積極的處世態度。同時也要感謝叔叔與姑姑長期對我的關心與照顧。

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The complexities of networks have been increasing as the networks are developing. Therefore, it requires simulation users to spend more and more time to simulate a large-scale and complex network case. Shortening the required time for simulating a large-scale and complex network has long been a desire. In the past, this goal could be easily achieved by simply using a faster CPU. However, as CPU clock speed increases become more and more difficult to achieve in recent years, this simple approach is no longer effective.

Understanding that it is harder and harder to increase CPU clock rate, major CPU vendors such as Intel, AMD, etc. have all turned to multicore CPUs as the best way to gain additional performance. Nowadays, there are many models of desktop PCs, notebook computers, and servers on the market that are already shipped with dual-core or quad-core CPUs. With this trend, it is very likely that after a few years, uni-core systems may disappear on the market.

Since multicore systems will be everywhere in the future, effectively using the computing power of multiple cores (We will use the more conventional term "CPU" to refer to "core" in the rest of the thesis when there is no ambiguity) simultaneously is a strong desire. However, as pointed out in [1], it is difficult for an application (including a network simulator) to automatically gain performance speedups on multicore systems because at any given time the application can only be run on a single CPU.

To gain performance speedups, an application needs to be made "multi-threaded" so that its threads can be run on multiple CPUs simultaneously. However, making an application program "multi-threaded" is not a simply task and does not necessarily can provide good performance speedups [1].

In the context of network simulation, many methods have been proposed for parallel and distributed simulations over multiple CPUs to achieve performance speedups [2]. Most of these methods fall into two approaches – the conservative approach and the optimistic approach. Using either approach requires a simulation user to partition a simulated network into several portions and modify simulation code to perform simulation clocks synchronization among these portions to avoid causality errors. The conservative approach, although simpler to be implemented into the simulation code, usually results in very low simulation performance under tiny lookahead values [2]. On the other hand, the optimistic approach, although potentially may achieve higher performance speedups than the conservative approach, is very complicated and requires substantial modifications to the simulation code.

Network simulation users want to enjoy performance speedups without changing the way they use existing sequential network simulators. If this can be done, even if the performance speedup is not much, as long as the speed of a parallel network simulation is no less than that of the corresponding sequential network simulation, there is no performance loss to a simulation user. With this worst-case performance guarantee and the advantage of using exactly the same way to conduct parallel simulations, simulation users will be more willing to conduct parallel simulations to achieve potential performance speedups on multicore systems.

We proposes a novel and general parallel simulation approach, called the "Event-level Parallelism (ELP) approach," to achieve the above goals. The idea of this approach is analogous to the idea of the "instruction-level parallelism" approach employed in the computer architecture and compiler research communities, which exploits instruction-level parallelism to achieve performance speedups over multiple CPUs without finding the parallelism inherent and latent in an application.

The rest of the thesis is organized as follows. In the Chapter 2, we present the

overview of parallel and distributed simulation methods and the state-of-the-art of the multi-threaded design in the Linux systems. The NS2 network simulator is also introduced in this chapter. The ELP approach is presented and discussed in Chapter 3. We describe the detailed design and implementation os the ELP approach in Chapter 4. Chapter 5 presents experimental results to evaluate the performance speedups. Finally, we propose possible extensions to our implementation in Chapter 6 and conclude in Chapter 7.

Chapter 2

Background

The event-level parallelism approach is similar to the parallel and distributed simulation approaches to execute a simulated case on multicore CPU systems. The challenge in the ELP approach is the same at conventional parallel and distributed approaches to execute logical processes (LPs) concurrently and generate correct simulation results. In addition to these, a multi-threaded application running on multiple CPUs needs the support of an operating system (OS) and a user-level thread library.

In this Chapter, the overview of parallel and distributed simulation and related work are introduced. In addition, we explain the state-of-the-art of the multi-threaded design in the Linux kernel and in the user-level library. Finally, the background knowledge of the NS2 network simulator is also introduced.

2.1 Related Work

The parallelization of network simulations has been studied for some time. In [3], the authors attempt to parallelize the widely used open source network simulator ns but could only support simple point-to-point links with static routing and UDP traffic. The supports for TCP connections, dynamic routing, and shared medium networks were not provided due to high complexities. In [4], the authors attempt to parallelize a widely used commercial network simulator, called OPNET ([5]), but could only support simple UDP and IP protocols. The supports for TCP connections and other

protocols were not provided due to the extensive used of global state, zero lookahead interactions, and pointer data structures in OPNET.

The second reason is that simulation users need to learn parallel simulation concepts and approaches to conduct parallel network simulations. This hinders users without such training from conducting parallel network simulations. In [6, 7], the authors use a federated approach to interconnect multiple ns simulation engines, each of which simulates a portion of the network, to together simulate a network. Although this approach reduces the required modification to ns simulation code, a user still needs to modify the original script usages of ns for parallel simulations. Even if an existing network simulator has been completely parallelized, a user still needs to learn the concepts of parallel simulation so that he/she can wisely partition a simulated network into several portions and map them onto available CPUs. Partitioning a large-scale network with a large number of nodes is a tedious work for the user. Worse yet, finding a load-balancing partition to achieve good performance speedups is very difficult. To achieve good load balancing, the partition decision cannot be made simply by considering the number of nodes assigned to each portion, Instead, the amount f packet traffic that may be generated and needs to be processed in each portion should also be considered. However, such information is hard to obtain and estimate at the time when a user needs to partition a network for parallel simulation.

2.2 Parallel and Distributed Simulation Overview

The parallel discrete event simulation is referred to as the execution of a discrete event simulation program on parallel and distributed simulators. The major issue of the parallel and distributed simulation is how to ensure that results of the parallel discrete event simulation on multiple CPU are correct. In a sequential execution paradigm, it is crucial that the simulation engine always selects the smallest time-stamped event from its event list as the one to be processed next. If an event with larger timestamp were executed before one with a smaller timestamp, this simulation may result in incorrect results. We call this type of errors “causality errors.”

A discrete event simulation, consisting of logical processes (LPs) that interact exclusively by exchanging timestamp messages, obeys the local causality constraint if and only if each LP processes events in non-decreasing timestamp order. However, adherence to this constraint is sufficient, although not always necessary, to guarantee that no causality errors occur. In other words, violating causality constraint may not always result in simulation errors. This is because two events within a single logical process may be independent of each other. In such a case, processing them without using the non-decreasing timestamp sequence does not lead to a causality error.

The synchronization mechanisms for parallel and distributed simulation fall into two approaches – the conservative approach and the optimistic approach. Using conservative approach, each logical process follows the local causality constraint and thus is blocked until it can be guaranteed that its (local and remote) events are safe to process. Events of each logical process are processed strictly in the non-decreasing timestamp order to avoid any causality errors. On the other hand, with the optimistic approach, events can be processed out of the non-decreasing timestamp order by using additional recovering mechanisms, such as the "rollback" [2].

The ELP approach can easily avoid the causality errors on multicore systems. The detail of the ELP approach will be presented in Chapter 3.

2.3 The State-of-the-art Multi-threaded Support

As mentioned previously, a multi-threaded application running on multiple CPUs needs the support of an operating system (OS) and a user-level thread library. We explain those components in the following section respectively.

2.3.1 The Thread Design in the Linux Kernel

A process is defined as follows [8]: a process is an instance of a program in execution. When a process is created, it is almost identical to its parent. It receives a logical copy of the parents address space and executes the same code as the parent. They

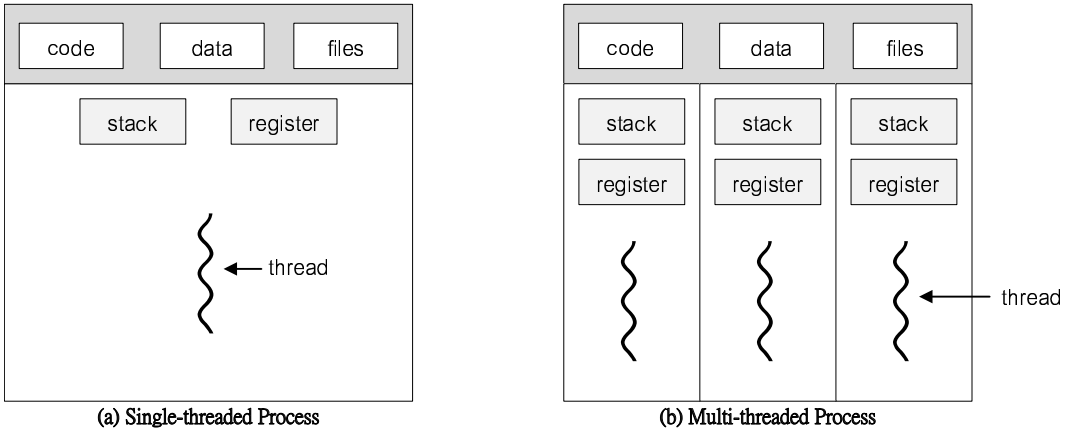


Figure 2.1: Thread and process models: (a) Single-threaded Process, and (b) Multi-threaded Process

have separate copies of the data (stack and heap). Changes by the child to a memory location are invisible to the parent, although the parent and child may share the memory pages containing the program code.

A thread is one of multiple execution flows of a process. It is a basic scheduling unit of CPU. There can be more than one thread in a process, and a process at least is associated with a thread. Therefore, the threads of a process may access the same address space and the same kernel resources. As illustrated in Fig. 2.1, each periphery block donates a process. As Fig. 2.1(a) shown, when there is only one thread associated with a process, the program is called a single-threaded application or a single-threaded program. On the other hand, there are more than one thread associated with a process in Fig. 2.1(b). We call this program a multi-threaded application or a multi-threaded program.

In most conventional operating systems, there are two types of thread. One is user thread and the other is kernel thread. A user thread is alive at the user-level. It is often created by the user-level library. On the other hand, a kernel thread is alive in the kernel-level.

In the earlier versions of the Linux operating system, the user threads of a multi-threaded application are created, handled, and scheduled entirely in the user-level library. All the user threads correspond to a process in the kernel. Therefore, when

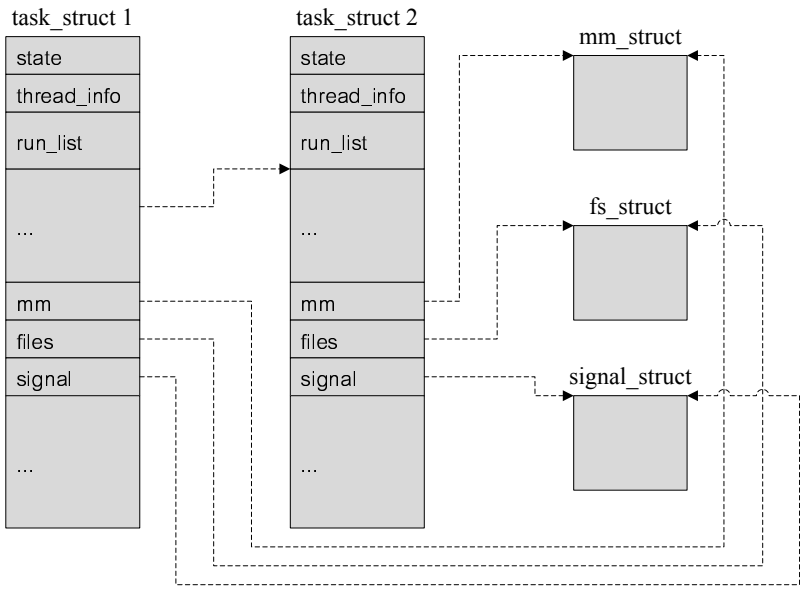


Figure 2.2: The architecture of the lightweight processes

one of the threads is accessing the Linux Kernel resources via a system call, the others are blocked by the Linux Kernel. Such a mechanism of multi-threaded support is not satisfactory.

A kernel thread is similar to a process and therefore it has many similar properties as a process. Kernel threads can be independently schedulable. Before the version-2.6 release of Linux system ([9]), the term "process" and "kernel-thread" are used interchangeably because there is no kernel thread design. Nowadays, the version-2.6 release of Linux kernel has supported the kernel thread. A kernel thread is also called a lightweight process (LWP) in the Linux system. When a process is an instance of a single-threaded program, the process just corresponds to a lightweight process, and the lightweight process is often called process.

The architecture of a lightweight process is illustrated in Fig. 2.2. As Fig. 2.2 illustrates, each lightweight process is associated with an independent task_struct. The task_struct is the most important data structure in the Linux kernel where the useful information of the lightweight process is stored. Some fields of the task_struct are pointers to a memory space where other used resources are stored. Lightweight processes and its parent process may share some resources in the kernel, such as the

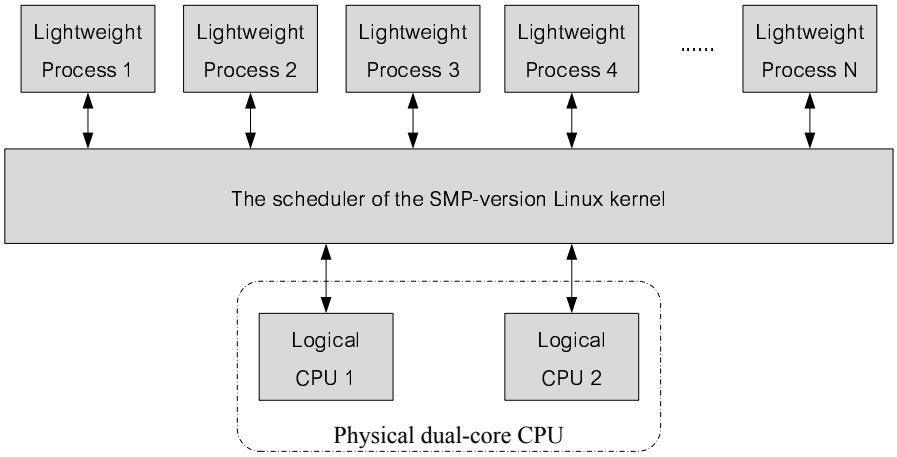


Figure 2.3: The architecture of the SMP-version Linux kernel over a dual-core CPU

memory page table, the file description table, etc. As such, when one of them modifies a shared resource, the other will immediately see the change. The Linux kernel must synchronize among them when they are accessing the shared resource.

A straightforward way to provide a better multi-threaded support is to associate a lightweight process (or a kernel thread) with a user thread. A Lightweight process can be scheduled. Therefore, each user thread can be independently scheduled by the Linux kernel.

The Linux operating system has supported the SMP architecture and integrated it since its version-2.6 release. Over an n-core CPU system, the SMP-version Linux kernel is capable of scheduling N lightweight processes on N CPU cores respectively. Therefore, the SMP-version Linux kernel is also capable of scheduling N user thread on N CPU cores simultaneously when each user thread is associated to a schedulable kernel thread. For example, as Fig. 2.3 depicts, there are two cores in the dual-core CPU. The scheduler is capable of scheduling two lightweight processes on two CPU cores simultaneously.

After the version-2.6.18 release, the Linux operating system has enabled the SMP support and the LWP support by default.

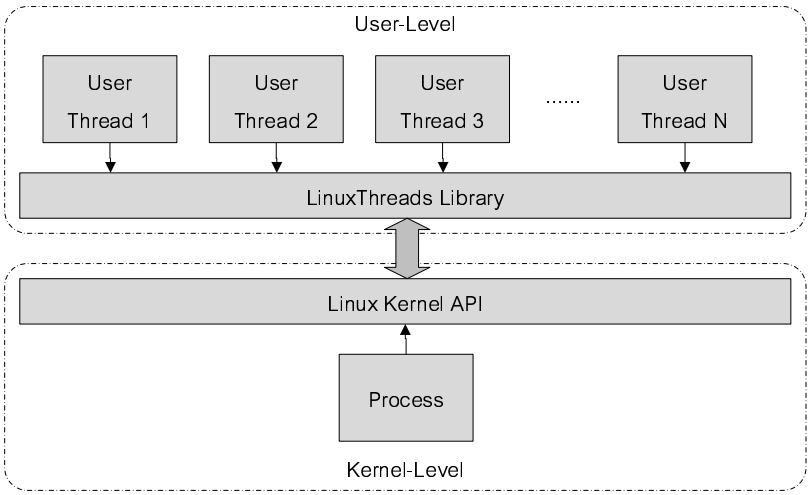


Figure 2.4: The model of the LinuxThreads library

2.3.2 The Thread Design in the User-level Library

The IEEE defines the POSIX 1003.1 standards ([10, 11]) to specify the application programming interface (API) for software compatible with variants of the Unix-like operating system, such as FreeBSD, Linux, Solaris, etc. Three well-known user-level libraries are developed.

The LinuxThreads Library

The first developed library is the LinuxThreads ([12]) library. Almost every conventional Linux distribution (a package containing not only a specified Linux system but also a lot of popular applications), such as Fedora Core, Ubuntu, Debain, etc. provides the LinuxThreads library to support multi-thread facilities.

As Fig. 2.4 illustrates, the LinuxThreads library implements Nx1 model: the all user threads of a multi-threaded application are associated with a process (or a light-weight process) in the Linux system. However, this architecture could not fully support the multi-threaded capability because it provides multi-threaded support entirely at the user-level. There are a number of problems with it, mainly owing to the implementation. For example, it uses the system call clone to create a new process sharing the parent's address space causing problems for signal handling; LinuxThreads needs

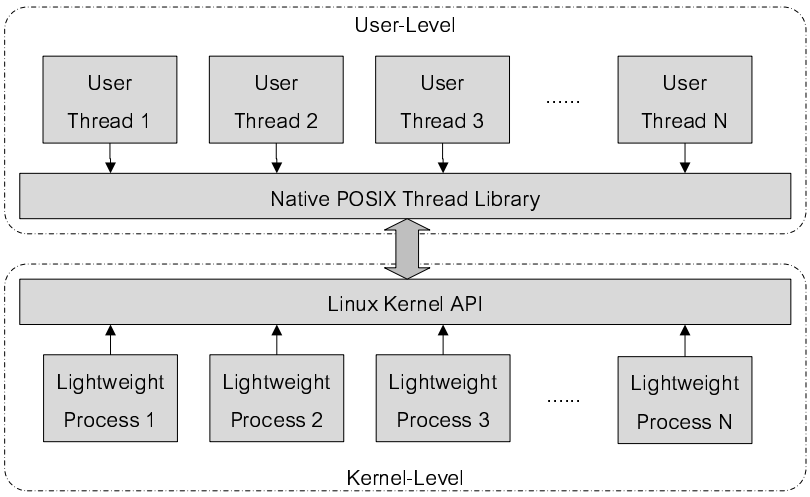


Figure 2.5: The 1x1 model of the NPTL library

to use the signal SIGUSER1 and SIGUSER2 for inter-thread communication, so that LinuxThreads must reserve those signals for it. In this model, only one of the user threads could access the kernel resource at any given time.

To improve upon LinuxThreads, rewriting the user threads library is required. Two projects were started to address these requirement: Next Generation POSIX Threads (NGPT) and Native POSIX Thread Library (NPTL). Since those new projects were started, the main maintainer of the LinuxThreads project has suspended the development of this project.

The NGPT Library

NGPT's main developer team included engineers from IBM, Inc. As the results presented in [13], the creating and destroying performance of NPTL is more efficient than NGPT. For this reason, the NGPT library was abandoned in mid-2003, at around the same time when NPTL was released, so that NPTL becomes the standard POSIX-compliant library in the Linux system. NPTL is now a fully integrated part of GNU C Library (it is also called glibc library).

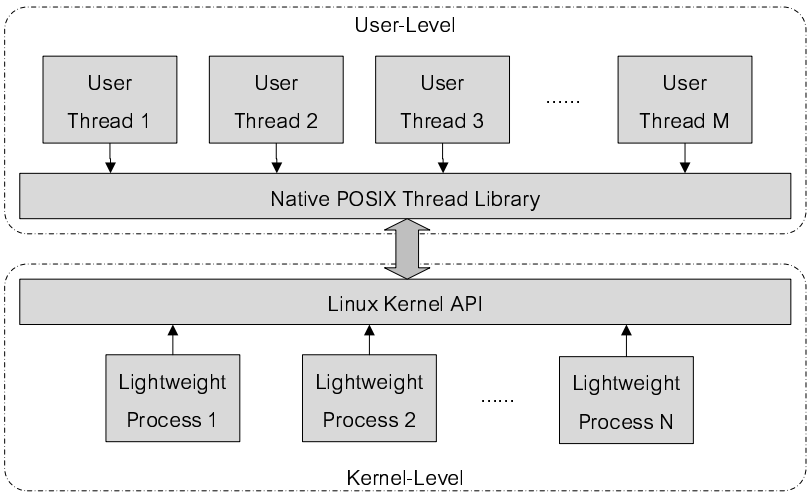


Figure 2.6: The MxN model of the NPTL library

The NPTL Library

NPTL ([13]) project was developed by a team included developer from Red Hat, Inc. They began to design almost at the same time as the NGPT project. It uses an approach similar to LinuxThreads that the NPTL library also uses the system call clone. However, the NPTL library uses the system call to create a new lightweight process rather than a process. NPTL requires a specialized kernel that provides the SMP and the LWP facilities to implement independently scheduling and sharing the address space among all kernel threads of a multi-threaded application. As aforementioned, the current version of the Linux kernel has implemented these requirements.

Two different implementation models are provided by the NPTL library. One model is the 1x1 architecture that each user threads is in 1-1 correspondent to a lightweight processes in the Linux kernel as Fig. 2.5 illustrats. This is the simplest implementation. In the 1x1 model, the scheduler of the Linux kernel may independently schedule all user threads of a multi-thread application on multiple CPUs, so that one user thread may sleep while others are running. They may also access the kernel resources concurrently.

The other model is the MxN architecture as Fig. 2.6 depicts. There are M user threads associated with N lightweight processes. The NPTL library needs to han-

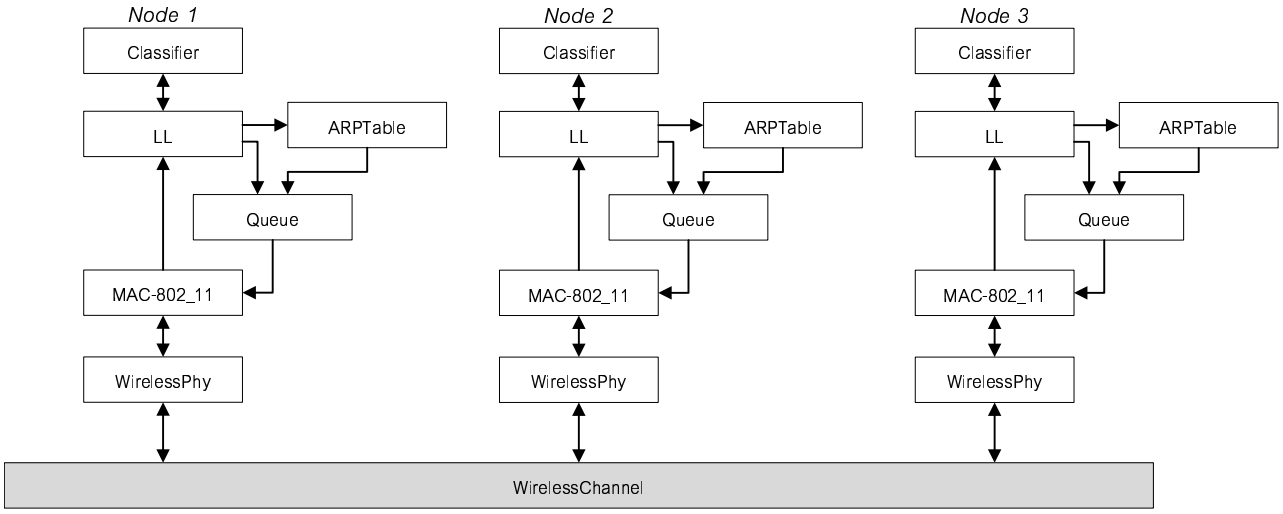


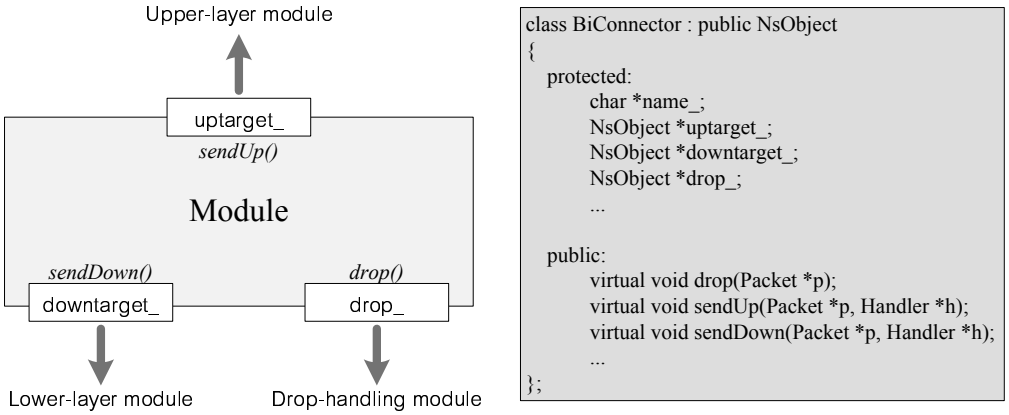
Figure 2.7: The module-based platform provided by the NS2 network simulator

dle, and schedule those user threads entirely at th user-level. NPTL is capable of associating a scheduled user thread with an available lightweight process (or a kernel thread). In other words, the MxN model of the NPTL library must make context switching of user threads in the library. The model increases the complexity of the implementation, and therefore there is no developer who is implementing this model nowadays. All Linux distributions have adopted the NPTL library with the 1x1 model to support multi-threaded applications.

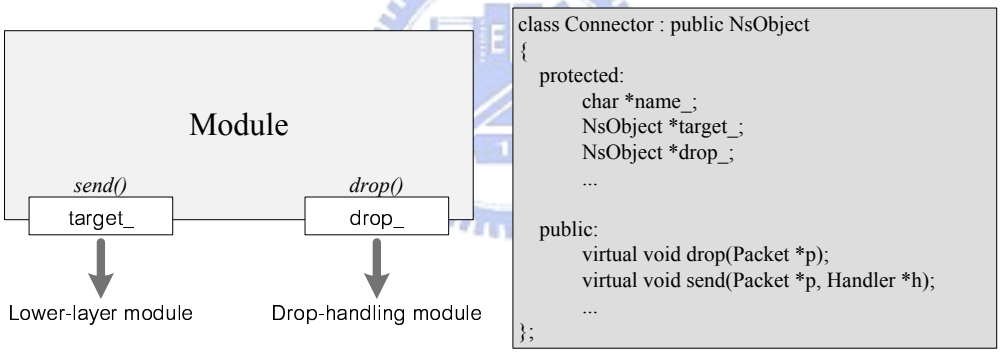
2.4 The NS2 Network Simulator

NS2 is an object-oriented and module-based simulator developed as part of the VINT project at the University of California in Berkeley. NS2 is a discrete event simulator targeted at networking research. It provides substantial support for simulation of TCP, UDP, routing protocols over wired and wireless networks. Its performance depends on the number of events that it needs to process. The more events it needs to process, the slower its simulation speed will be.

It adopts an open-system architecture and provides a module-based platform. Fig. 2.7 is an example that depicts a wireless network topology consisting of three



(a) The connector-module skeleton



(a) The bi-connector-module skeleton

Figure 2.8: The module skeleton provided by the NS2 network simulator

nodes and the organization of each node. In the module-based platform, a module skeleton is provided as shown in Fig. 2.8. Based on the skeleton, researcher can easily develop their own modules and integrate them into the simulator.

Chapter 3

The Event-level Parallelism Approach

3.1 The ELP Architecture Overview

In this chapter, we present the architecture of the ELP approach. Fig. 3.1 shows the architecture of a parallel network simulator using the ELP approach. For illustration purposes, the parallel network simulator depicted in this figure is for dual-core systems. It can be extended for quad-core systems by simply increasing the number of worker threads from two to four in this figure.

There are four important components of the ELP architecture. In the following, we describe those components respectively.

- **Master Thread**

  The major responsibility of the master thread is to determine currently which events are safe to be executed in parallel without causing causality errors among themselves

- **Worker Thread**

  A worker thread only executes safe events which have been determined by master thread as safe events and placed in the safe event list. It may insert the newly

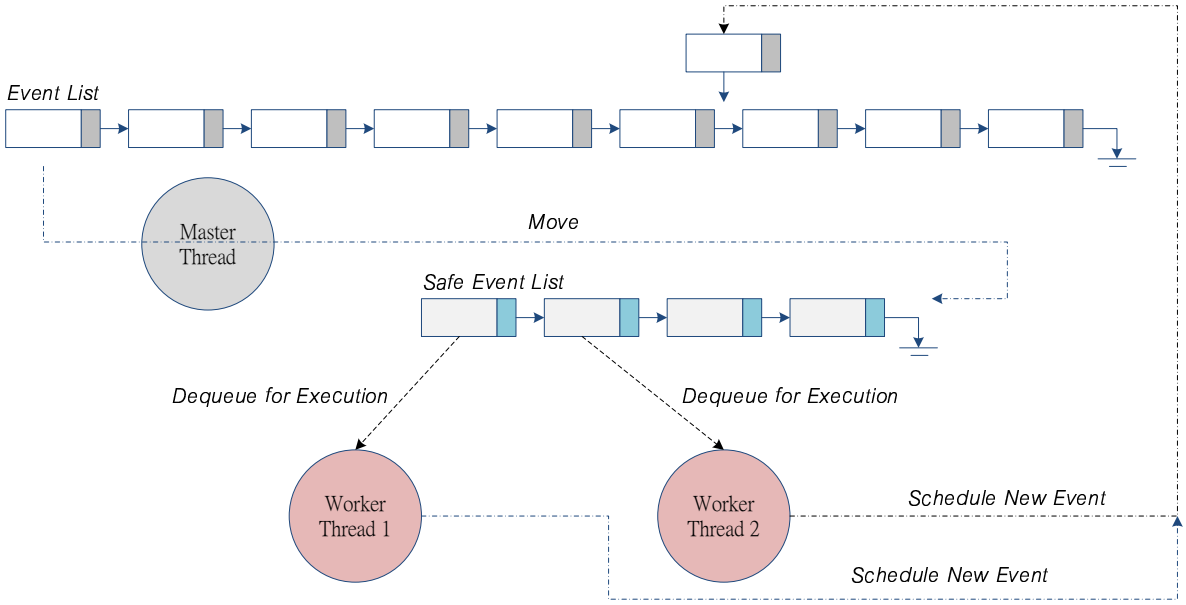


Figure 3.1: The architecture of a parallel network simulator using the ELP approach for dual-core systems

generated events into the event list during event execution.

- **Event List**

  All newly generated events are stored in the event list. The event list is accessed by master or worker threads.

- **Safe Event List**

  The safe event list stores the safe events to be executed in the future. Both master and worker threads can access this list.

For a dual-core, only three simple threads need to run in the parallel network simulator. One thread is the master thread while the others are two worker threads. The master thread determines currently which events are "safe" [2] to be executed by worker threads in parallel without causing causality errors among themselves. When such events can be found, the master thread moves them from the event list to the safe event list and wakes up any worker thread that is sleeping waiting for a safe event to execute. As Fig. 3.2 illustrats, if no new safe events can be found at the current time,

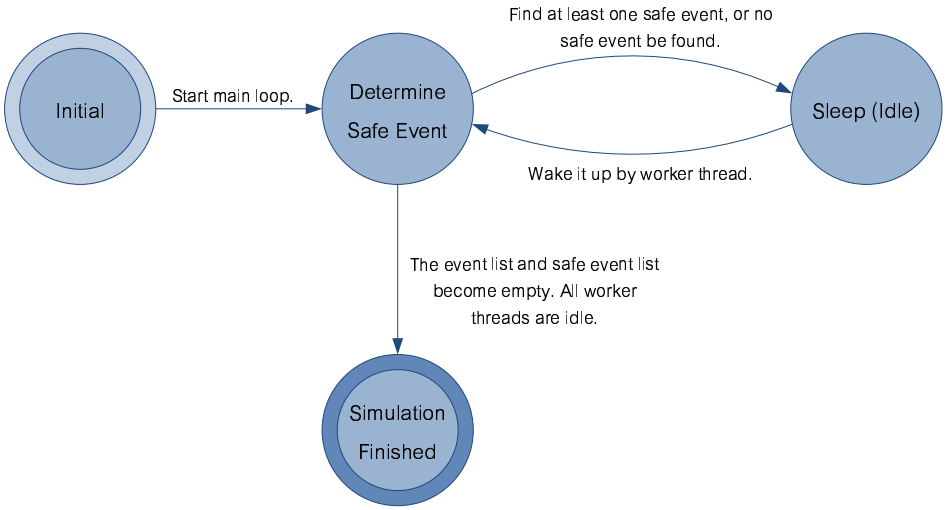


Figure 3.2: State translation diagram of the master thread

the master thread sleeps and waits for any worker thread to later wake it up when that worker thread has finished its current event computation. After being waked up, the master thread continues to find new safe events. The master thread repeats the above operations until the whole simulation is finished (i.e., when the event list becomes empty).

At any given time, at least one worker thread must be busy executing an event. The ELP approach has this property because as in a sequential simulation, the event with the smallest timestamp in the whole simulated network must be a safe event for execution. As such, suppose that currently there is only one worker thread busy executing an event, when that worker thread has finished its event computation and become idle because there is no safe event in the safe event list for it to execute, the master thread must be able to find at least one safe event (it is simply the event at the head of the event list) and move it (or them) to the safe event list. At that time, one or more worker threads will have safe events to execute again as Fig. 3.3 depicts. With this property, the master thread is assured that, when all safe events have been processed, it will be waked up by one worker thread to continue to find more safe events from the event list. It is also this property why the ELP approach can always generate a performance speedup higher than or close to 1 even under tiny

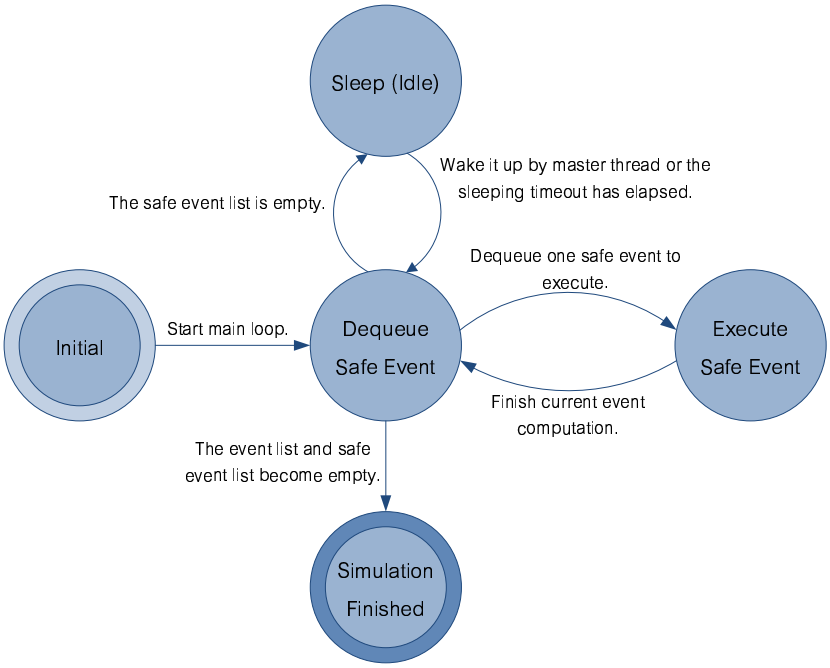


Figure 3.3: State translation diagram of the worker thread

lookahead values. This is because under such a harsh condition, the ELP approach can always degenerate to a sequential simulation approach. This can be simply done by activating only one worker thread and asking the master thread to move the first event in the event list to the safe event list without checking whether it is a safe event. As such, the ELP approach will never result in a parallel simulation that runs much slower than a sequential simulation.

The event list stores the events to be executed in the future and sorts them based on their timestamps, which denote the times when the events should be executed, in a non-decreasing order. In a sequential network simulator running on a single CPU, these events will be dequeued and executed sequentially by the CPU to avoid causality errors. If new events are created (called "scheduled" in the simulation research field) when an event is executed by a worker thread, the newly generated events are inserted into the event list based on their timestamps.

In the ELP approach, two worker threads are created on a dual-core system to fully utilize the two CPUs. Found safe event are stored and sorted in the safe event list, from which any worker thread can dequeue a safe event to execute it when that

worker thread finishes its current event computation and becomes idle.

As long as the two worker threads are busy executing events at all times, the two CPUs will be fully utilized and good performance speedups will result. The important job of the master thread is to find enough ELP for the target N-core system at all times. For a N-core system, maintaining N safe events in the safe event list at all times suffices to keep the N worker threads busy all the time.

When a worker thread finishes its current event computation, wants to dequeue a safe event from the safe event list, but finds that there is no safe event in the safe event list, it wakes up the master thread and asks it to find more safe events and move them into the safe event list, If more safe events can be found, the master thread will wake up this worker thread and this worker thread will have a safe event to execute and become busy again. If no more safe events can be found (e.g., the other worker thread is executing an event, which can affect the events at the front of the event list), this worker thread will sleep, waiting for the master thread to later wake it up when the master thread has found new safe events and inserted them into the safe event list.

Since the master thread and worker threads all need to access the event list and the safe event list, to ensure data structure consistency, accesses to these lists are coordinated and protected by locks. In addition to these two lists, if there are other global variables or data structures that may be accessed during event execution, they need to be protected by locks as well. One global variable that is accessed very frequently during event execution is the variable storing the current simulation time. However, because this variable is accessed only for read purposes during event execution and its value can only be advanced by the event-processing loop of the network simulator (in the master thread), it need not be protected by locks. When a worker thread dequeues a safe event from the safe event list for execution, the timestamp of the event to be executed should be the current simulation time for this specific event. As such, one can store it into a local variable of this worker thread. When the worker thread executes this event and needs to get the current simulation time by calling an API function, this API function can retrieve the value of this local variable based on

the thread ID and return it as the current simulation time for this specific event. No locking/unlocking overhead is needed for accessing the current simulation time of a simulated network.

3.2 The Affect Event Relationship

In this section, we present how to find safe events for parallel execution by worker threads. The number of safe events that can be found strongly depends on the lookahead value. Therefore, we briefly explain the concepts of lookahead and safe event here. More detailed explanations are presented in [2]. Suppose that a sequential network simulator has advanced its simulation clock to simulation time T, which is the timestamp of the next unprocessed (i.e., the first) event in the event list, and is going to execute that event. Suppose that there is a constraint that a new event must be scheduled at least L units of simulation time into the future, then it can be guaranteed that all new events scheduled during the execution of this event must have a timestamp greater than or equal to $T + L$. With this property, any event in the event list with a timestamp less than $T + L$ can be safely processed without causing causality errors. These events are called "safe events" and L is the lookahead value for this example.

In the context of network simulation, when a packet is transmitted over a link, one can associate a lookahead value with the link. This value is the sum of the signal propagation delay D over the link and the transmission time $TxTime$ of the packet being transmitted over the link, where D is a fixed value and $TxTime$ is the packet length divided by the link bandwidth. This is because when a packet is transmitted over a link, only then this amount of time has elapsed will the packet arrive at the other end of the link and be completely received by the remote network interface. Suppose that the current simulation time is $T1$ and an event $E1$ is executed at the source node of a link, and during the event execution a packet is transmitted to the destination node of the link. As Fig. 3.4(a) depicts, if there is an event $E2$ in the event list with a timestamp of $T2$ larger than $T1$, one can be sure that if $T1 + D + TxTime > T2$,

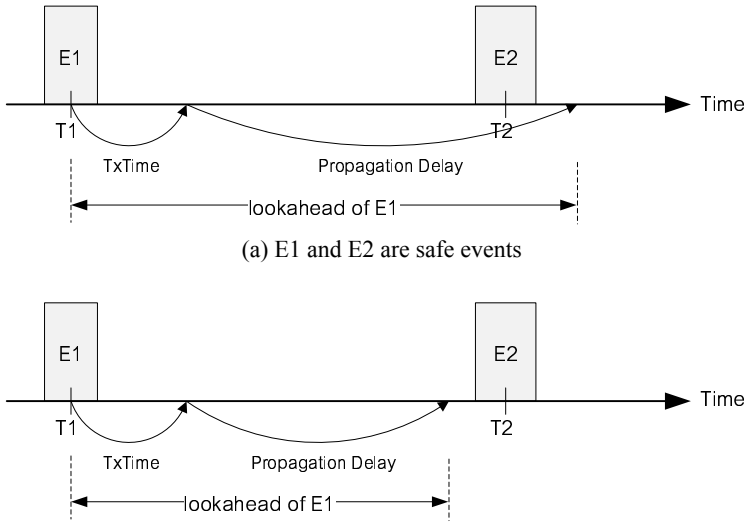
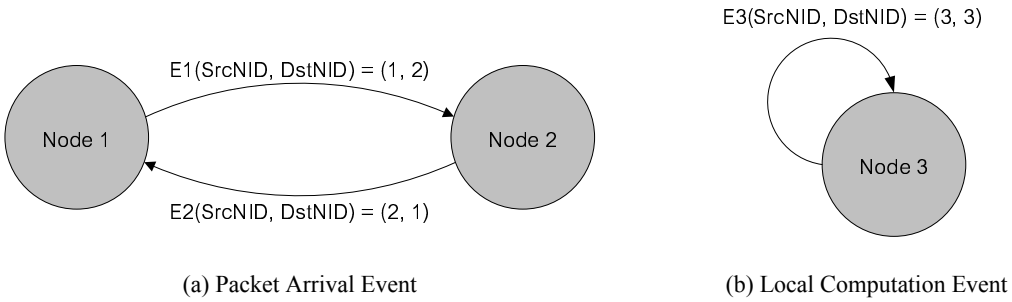


Figure 3.4: Lookahead value is amount of transmission time plus propagation delay

$E1$ cannot affect $E2$. This means that in this condition $E1$ and $E2$ are safe events and can be executed in parallel without affecting each other. On the other hand, if $T1 + D + TxTime \leq T2$ as Fig. 3.4(b) illustrats, $E1$ can affect $E2$ because the arrived packet may change the internal state of the node where $E2$ resides before $E2$ is executed. In such a condition, only $E1$ is a safe event while $E2$ is not.

When two nodes are not directly connected by a link, an earlier event $E1$ on one node may or may not affect a later event on the other node. In the following, we present the rules used to check whether it is possible for $E1$ to affect $E2$. In the ELP approach, the data structure of each event stores the source node ID (SrcNID) and destination node ID (DstNID) of the event. If the event represents a packet arrival sent from node i to node j, then SrcNID is set to i and DstNID is set to j. A packet arrival event, when executed, will change the internal state (e.g., the buffer occupancy level) of the receiving node. On the other hand, if the event does not represent a packet arrival, it must represent a local computation, which only modifies the internal state of the node where this event is executed and does not transmit a packet to another node. For a local computation event, suppose that the event is to be executed on node k, then both of its SrcNID and DstNID are set to k. For example, as



(a) Packet Arrival Event (b) Local Computation Event

Figure 3.5: Two typical events: (a) Packet Arrival Event, and (b) Local Computation Event

Fig. 3.5 depicts, there are three events in the simulated case currently. One is a local computation event and the others are arrival events. The local computation event E3 resides on Node 3 so that its SrcNID and DstNID are set to 3. The two arrival events E1 and E2 are sent from Node 1 to Node2 and from Node 2 to Node 1, respectively. The SrcNID and DstNID of the event E1 are set to 1 and 2, those of the event E2 are set to 2 and 1. The The SrcNID and DstNID information of an event are readily available in a network simulator.

In a protocol stack consisting of several protocol modules, a packet arrival event for the destination node of a link can only be scheduled by the bottom physical-layer protocol module, which models the operation of the network interface at the source node of the link. Consider the triggering of a "hello" timer, which is commonly used in a routing protocol module to send out a HELLO packet to inform other nodes that this node is still alive. In this example, although the upper-layer routing protocol module schedules a packet transmission event (not an arrival event), before this event reaches the bottom physical-layer protocol module, as it passes along all protocol modules, it is still considered a local computation event because the execution of this event only affects the internal state of the local node.

In the following, we discuss whether an event on one node can affect an event on another node.

Table 3.1: Four different conditions of regarding SrcNID and DstNID

Rules	**Condition**
Rules 1	$SrcNID_1 \neq SrcNID_2$ and $DstNID_1 \neq DstNID_2$
Rules 2	$SrcNID_1 = SrcNID_2$ and $DstNID_1 \neq DstNID_2$
Rules 3	$SrcNID_1 \neq SrcNID_2$ and $DstNID_1 = DstNID_2$
Rules 4	$SrcNID_1 = SrcNID_2$ and $DstNID_1 = DstNID_2$

3.2.1 Packet Arrival Events

In this section, we assume that the two events considered are both packet arrival events. Denote E1's SrcNID and DstNID as $SrcNID_1$ and $DstNID_1$ and E2's SrcNID and DstNID as $SrcNID_2$ and $DstNID_2$. When one compares $SrcNID_1$ with $SrcNID_2$ and compares $DstNID_1$ with $DstNID_2$, based on the two comparison results (same or not), there are four different conditions to consider shown in Table 3.1. Among these conditions, we consider Rule 1 and Rule 2 preliminary safe conditions – meaning that if certain lookahead conditions can be met, it is very likely that $E1$ cannot affect $E2$. As such, to avoid wasting CPU cycles, one can simply view that $E1$ cannot affect $E2$ without further checking the lookahead conditions.

The reasons for the above checking rules are explained below in Fig. 3.6. In this figure, each node has a name N_i, where i denotes the ID of the node and each link has a delay denoted as L_ij, where i and j are the IDs of the two nodes of the link. Each node connects to each of its immediate neighbors by a link and each ink has an output buffer associated with it. In this figure, Ea, Eb, Ec, Ed, and Ef represent packet arrival events and Ee represents a local computation event. Each packet arrival event is associated with an arrow link denoting its $SrcNID$ and $DstNID$. For example, the $SrcNID$ and $DstNID$ of Ea are 6 and 5, respectively, and those of Ed are 8 and 7, respectively. On the other hand, each local computation event is associated with a self-pointing arrow and its $SrcNID$ and $DstNID$ are the same and is the ID of the node where this event is to be executed. For example, the $SrcNID$ and $DstNID$ of Ee are the same and is 4.

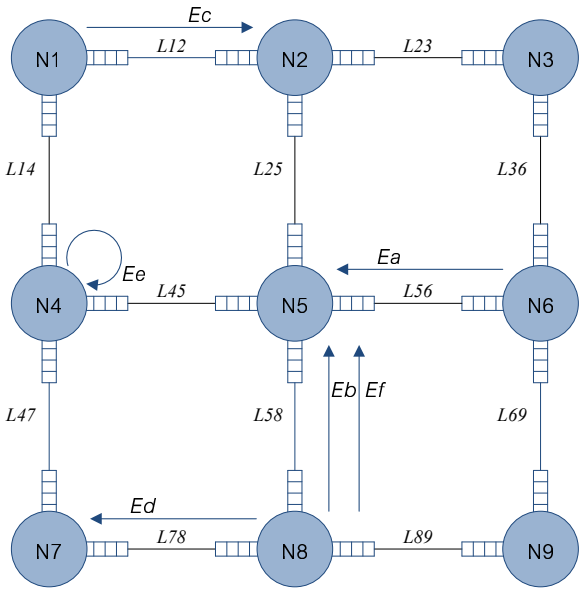


Figure 3.6: A 3x3 grid wired network used to illustrate safe events found at the event level

The condition of Rule 3 that $SrcNID_1 \neq SrcNID_2$ and $DstNID_1 = DstNID_2$ is considered unsafe. An example for this condition is when one considers whether Ea can affect Eb assuming that the timestamp of Ea is smaller than that of Eb. Since a packet arrival can change the internal state of the receiving node, it is clear that Ea can affect Eb and thus the execution order of Ea and Eb should be maintained. As such, Ea and Eb should not be executed in parallel.

The condition of Rule 2 that $SrcNID_1 = SrcNID_2$ and $DstNID_1 \neq DstNID_2$ is considered safe if certain lookahead condition can be met. An example for this condition is when one considers whether Eb can affect Ed assuming that the timestamp of Eb is smaller than that of Ed. Although the destination node of Eb, which is $N5$, is not the same as the destination node of Ed, which is $N7$, Eb can still affect Ed if there exists a path from $N5$ to $N7$. On the other hand, if there is no path from $N5$ to $N7$, one can be sure that Eb cannot affect Ed.

To determine whether an earlier event $E1$ on a node Src may affect a later event $E2$ on a node Dst, one needs to compute the minimum path lookahead among all possible paths from Src to Dst. The path lookahead of a path $Src \rightarrow N_i \rightarrow N_j \rightarrow ... \rightarrow N_k \rightarrow$

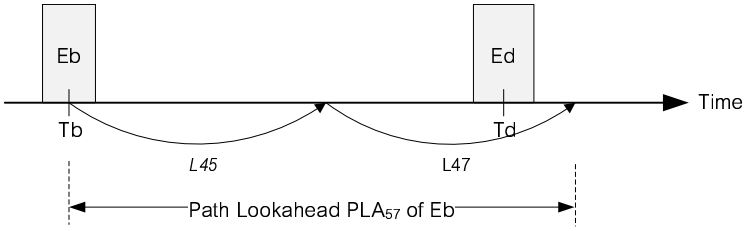


Figure 3.7: The path lookahead of a path is the sum of the link lookaheads of all links on the path

Dst is defined to be the sum of the link lookaheads of all links on the path. The computed minimum value represents the minimum amount of time that must elapse before an earlier event on Src can affect a later event on Dst. This is because Src may schedule (create) an event destined for N_i, upon receiving this event, node N_i may in turn schedule an event destined for N_j, ... , and N_k may in turn schedule an event destined for Dst. If the event scheduled by N_k has a smaller timestamp than $E2$ and thus gets executed prior to $E2$ on Dst, $E1$ can affect $E2$.

For the above case in which one would like to determine whether Eb on N_5 may affect Ed on N_7, one needs to compute the minimum path lookahead among all possible paths from N_5 to N_7. One can use the all-pairs shortest path Dijkstra's algorithm to precompute and store the minimum path lookaheads between all pairs of nodes. Suppose that the precomputed minimum path lookahead of the path from N_5 to N_7 is PLA_{57} and, as Fig. 3.7 depicts, the timestamp of Eb is Tb and that of Ed is Td. Then if $Tb + PLA_{57} > Td$, Eb cannot affect Ed and they can be executed in parallel.

The condition of Rule 1 that $SrcNID_1 \neq SrcNID_2$ and $DstNID_1 \neq DstNID_2$ is considered safe and the treatments for it are the same as those for the previous condition in which $SrcNID_1 = SrcNID_2$. An example for this condition is when one considers whether Ec can affect Ed assuming that the timestamp of Ec is smaller than that of Ed. Like the treatments for the previous condition, suppose that the precomputed minimum path lookahead of the path from N_2 to N_7 is PLA_{27} and the timestamp of Ec is Tc and that of Ed is Td. Then if $Tc + PLA_{27} > Td$, Ec cannot

affect Ed and they can be executed in parallel.

The condition of Rule 4 that $SrcNID_1 = SrcNID_2$ and $DstNID_1 = DstNID_2$ is considered unsafe. An example for this condition is when one considers whether Eb can affect Ef assuming that the timestamp of Eb is smaller than that of Ef. Although a network interface can handle only one packet transmission at one time, it is possible that many packet arrival events with the same $SrcNID$ and the same $DstNID$ (i.e., they are transmitted over the same link) are scheduled and appear in the event list. For example, suppose that a link has a long delay and a high bandwidth, which makes the transmission time of a packet over this link less than the link delay. Then several transmitted packets may be simultaneously "on the flight" over this link. It is clear that the order of these packet arrivals at the receiving node should be maintained and not be executed in parallel. Otherwise, causality errors will result.

3.2.2 Local Computation Events

Here we consider whether an earlier $E1$ on one node may or may not affect a later event $E2$ on another node, where either $E1$ or $E2$ or both are local computation events. Recall that for a local computation event, its $SrcNID$ is the same as its $DstNID$. Therefore, when discussing a local computation event, it suffices to only consider its $SrcNID$ or its $DstNID$.

Like the treatments for the previous subsection, when either $E1$ or $E2$ or both are local computation events, the conditions that $DstNID_1 = DstNID_2$ is considered unsafe. We only discuss the conditions that $DstNID_1 \neq DstNID_2$ using Fig. 3.8 that is similar to Fig. 3.6. In this figure, Ea and Ec represent packet arrival events and Eb and Ed represent local computation events.

Suppose that both $E1$ and $E2$ are local computation events, it is easy to decide whether $E1$ can affect $E2$. If $DstNID_1 = DstNID_2$, $E1$ can affect $E2$ because they are executed on the same node. On the other hand, if $DstNID_1 \neq DstNID_2$, $E1$ can still affect $E2$ through a path from node $DstNID_1$ to node $DstNID_2$. An example for this situation is when one considers whether Eb can affect Ed assuming that the

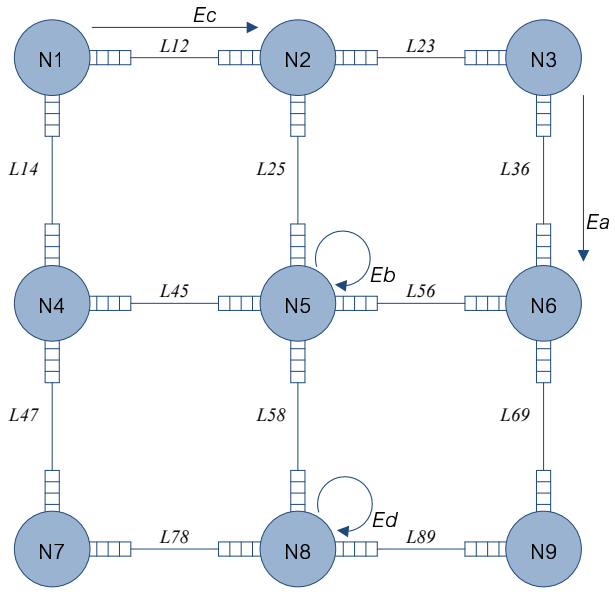


Figure 3.8: A 3x3 grid wired network used to illustrate safe events found for local computation events at the event level

timestamp of Eb is smaller than that of Ed. Although the destination node of Eb, which is N_5, is not the same as the destination node of Ed, which is N_8, Eb can still affect Ed if there exists a path from N_5 to N_8. On the other hand, if there is no path from N_5 to N_8, one can be sure that Eb cannot affect Ed. Link the previous treatments, one needs to compute the minimum path lookahead from node N_5 to node N_8. Let it be denoted as PLA_{78}. If the timestamp of Eb plus PLA_{78} is greater than the timestamp of Ed, Eb cannot affect Ed; otherwise, Ea can affect Eb.

So far, the lookaheads we have considered all come from the delays of physical links. However, it is possible that a local computation event can have a non-zero lookahead due to specific protocol designs. In such a situation, even if $E1$ and $E2$ are on the same node, if the timestamp of $E1$ plus its lookahead is greater than that of $E2$, $E1$ cannot affect $E2$.

For the case when $E1$ is a local computation event while $E2$ is a packet arrival, if $DstNID_1 \neq DstNID_2$, $E1$ is possible to affect $E2$. An example for this situation is when one considers whether Eb can affect Ec assuming that the timestamp of Eb is smaller than that of Ec. The destination node of Eb is N_5 and that of Ec, which is

N_2. Like the treatments for the previous situation, suppose that the minimum path lookahead of the path from N_5 to N_2 is PLA_{52}. If the timestamp of Eb plus PLA_{52} is greater than the timestamp of Ec, Eb cannot affect Ec; otherwise, Eb can affect Ec.

For the case when $E1$ is a packet arrival event while $E2$ is a local computation event, if $DstNID_1 \neq DstNID_2$, $E1$ may affect $E2$. An example for this situation is when one considers whether Ea can affect Eb assuming that the timestamp of Ea is smaller than that of Eb. The destination node of Ea is N_6 and that of Eb is N_5. In this situation, the minimum path lookahead PLA_{65} is computed. If the timestamp of Ea plus PLA_{65} is greater than the timestamp of Eb, Ea cannot affect Eb; otherwise, Ea can affect Eb.

3.2.3 Wireless Mobile Networks

The links we have considered so far are full-duplex fixed links. On such links, packet transmissions over different links are independent and do not affect each other. On a wireless channel, however, a transmitted packet is broadcast to all nodes that can sense the signal of the packet. As such, the network simulator needs to schedule a packet arrival event for every node that is within the interference range of the transmitting node.

To apply the ELP approach to a wireless network such as a mobile ad hoc network, because link and path lookahead information are important for the ELP approach, the network simulator first uses the wireless interface's interference range to construct the network topology. This is done by checking whether node j is within the interference range of node i. If this condition holds, it is considered that there is a wireless link from node i to node j. The lookahead of such a link is the link delay (which is the signal propagation delay from node i to node j) plus the packet transmission time over the link. After the network topology has been constructed, a wireless network can be viewed as a wired network and the checking rules for wired networks can be used for the wireless network. For a mobile ad hoc network, since nodes may move, the network topology constructed at the beginning of the simulation needs to be updated

during the simulation. The update frequency depends on the maximum moving speed of mobile nodes and the required accuracy of simulation results. The mobile ad hoc network will not be discussed in this thesis.

3.3 The Safe Event Set

In the following, we present how the master thread finds the safe event set and when it moves these events from the event list to the safe event list for execution by worker threads. In Fig. 3.9, we show a snapshot of the event list, the safe event list, and the two worker threads. If there is an arrow from event Ei to event Ej, this means that the master thread has determined that Ei cannot affect Ej. On the other hand, if there is no arrow from event Ei to event Ej, this means that Ei can affect Ej. As shown in this figure, for discussion purposes we name the events stored in the event list, starting from the head, $E1$, $E2$, $E3$, $E4$, ..., respectively. The master thread tries to find a safe event set from these events and move them to the safe event list. To do so, for $E1$, it checks whether $E1$ can affect $E2$, $E3$, $E4$, $E5$, $E6$, ..., $E(1+MP)$. For $E2$, it checks whether $E2$ can affect $E3$, $E4$, $E5$, $E6$, $E7$, ..., $E(2+MP)$. That is, for Ei, it checks whether $E(i)$ can affect $E(i+1)$, $E(i+2)$, $E(i+3)$, $E(i+4)$, ..., $E(i+MP)$, where MP is a system parameter (standing for "Maximum Parallelism") and depends on the degree of ELP that one would like to find. The value of MP also regulates the maximum number of events that the master thread checks for inclusion into the safe event set. Using an appropriate value for MP can provide adequate ELP for a N-core system while reducing unnecessary wastes of CPU cycles, which is explained below.

Suppose that one would like to maintain N safe events in the safe event list at all times, then MP should be set to a value no less than N. For a N-core system, although setting MP to N can potentially allow every worker thread to have an event to execute at the same time, because the number of events that can be included into a safe event set may be less than N (see below) and to avoid frequent sleep/wakeup interactions between worker threads and the master thread, it is suggested that MP is set to a

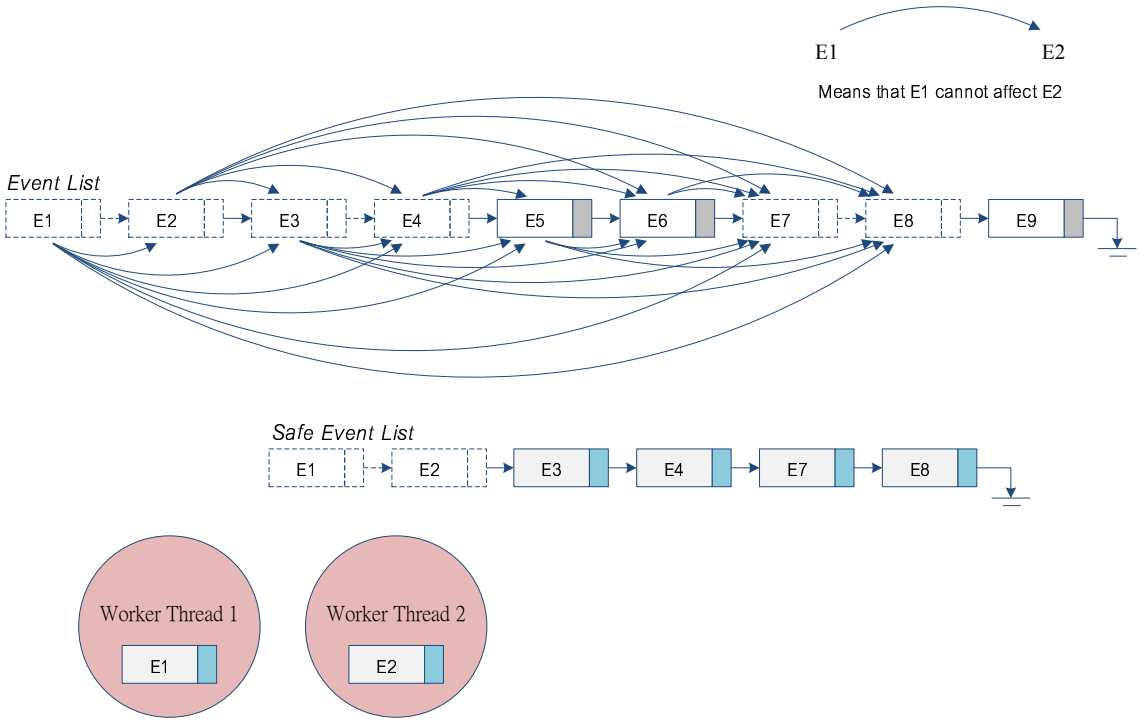


Figure 3.9: Determining when an event can be moved from the event list to the safe event list for execution by worker threads

larger value such as $2 * N$. On the other hand, MP should not be set to a too large value. Otherwise, excessive CPU cycles will be wasted on many unnecessary "affect" relationship comparisons, reducing available CPU cycles that can otherwise be used to achieve higher performance speedups. Recall that the master thread constantly determines the "affect" relationships among MP events and this operation requires $O(MP^2)$ time complexity. As such, if the N CPUs of a N-core system are already being fully utilized, further increasing MP to find more ELP will not improve the achieved performance speedup but may instead harm it.

A safe event set is constructed based on the above computed "affect" event relationships. Initially, it is an empty set {}. Starting from the first event $E1$ in the event list, event by event, one sequentially determines whether an event $E(i)$ can be added to the safe event set. If any of the events in the safe event set can affect $E(i)$, $E(i)$ cannot be added to the set. On the other hand, to include $E(i)$ into the safe event set, one needs to further check whether any of the events $E1$, $E2$, $E3$, $E4$, ...,

$E(i-2)$, and $E(i-1)$ that are not included in the current safe event set can affect $E(i)$. If none of them can affect $E(i)$, then $E(i)$ can be added to the safe event set. The following is a step-by-step construction of the safe event set for the event list in Fig. 3.9: $\{\}$, $\{E1\}$, $\{E1, E2\}$, $\{E1, E2, E3\}$, $\{E1, E2, E3, E4\}$, $\{E1, E2, E3, E4, E7\}$, $\{E1, E2, E3, E4, E7, E8\}$. The reason why $E5$ cannot be added to the safe event set is because $E2$ can affect it. The reason why $E6$ cannot be added to the safe event set is because $E1$ can affect it. In this figure, we use dash lines to draw $E1$, $E2$, $E3$, $E4$, $E7$, and $E8$ in the event list to represent the fact that they have been moved to the safe event list.

When an event stays in the safe event list, any available worker thread can dequeue it for execution. In Fig. 3.9, worker thread 1 has dequeued $E1$ and worker thread 2 has dequeued $E2$. We use dash lines to draw $E1$ and $E2$ in the safe event list to represent the fact that they have been dequeued by some worker threads and are still being processed. When a worker thread finishes its event computation, it will try to dequeue another event from the safe event list. If no safe event can be found, it wakes up the master thread asking the master thread to search for more safe events and move them from the event list into the safe event list. The master thread then uses the method described above to determine a new safe event set under the current condition (e.g., $E1$ is finished and removed from the safe event set). If new events can be added to the safe event set, then the master thread moves them into the safe event list. As shown in this example, events in the event list need not be sequentially moved into the safe event list. In this example, even though $E7$ and $E8$ have been moved into the safe event list, $E5$ and $E6$ still stay in the event list. This situation is correct because one had confirmed that $E5$ and $E6$ cannot affect $E7$ and $E8$ before adding $E7$ and $E8$ into the safe event set. Later on, when $E1$ and $E2$ are finished and have been removed from the safe event set, $E5$ and $E6$ will become qualified to be added to the safe event list.

When a new event E' is inserted into the event list (e.g., worker thread 1 may schedule this event when executing $E1$), one needs to check whether this event can be moved into the safe event list. Whether or not it can be moved into the safe

event list does not affect the events already in the safe event list. This is because all possible situation (i.e., $E1$ schedules E' and the execution of E' may affect an event in the safe event list) had already been considered when determining the safe event set. To determine whether E' can be moved into the safe event list, one uses the above method to construct a new safe event set based on the new event list. If E' appears in the new safe event set, it is moved into the safe event list.

Chapter 4

Design and Implementation

This chapter presents how to apply the ELP approach to the NS2 network simulator. We first describe the modification to the NS2 simulation engine and then the detailed design and implementation of the master thread, the worker thread, and the thread IPC. The modification of the wired and wireless modules on top of the NS2 network simulator is also presented.

4.1 The NS2 Simulation Engine Modification

The NS2 network simulator consists of C++ modules and TCL modules. A part of the information which is used during simulation is stored in C++ modules, and the other part of those is stored in TCL modules. To apply the ELP approach to the NS2 network simulator, some of these C++ and TCL modules need to be modified. In the following, we present the modified components of NS2 Simulation Engine, which the ELP approach depends on.

4.1.1 Scheduler

The NS2 network simulator uses the TCL description language to specify a simulated case. At the beginning of the simulation, the NS2 network simulator will first read this simulated case and configure all simulation environments. Then, the NS2 network

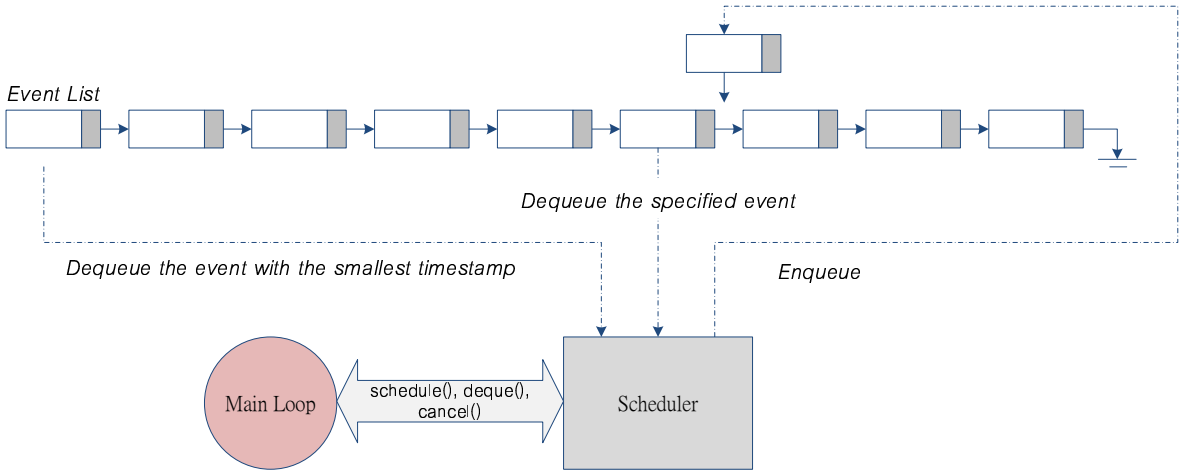


Figure 4.1: The relationship between the event list and the scheduler in the original architecture of the NS2 network simulator

simulator asks its scheduler to begin to simulate this case (it calls the function run() of the scheduler to perform this simulation run). It dequeues the event with the smallest timestamp from the event list and executes it sequentially before we apply the ELP approach. Scheduler repeats the above operations until the whole simulation is finished (when the event becomes empty) or a special event is executed, which is used to ask simulation engine to stop simulation.

To Apply the ELP approach to the NS2 network simulator, the scheduler of the NS2 network simulator needs to precompute the path lookahead, create the worker threads, and prepare the event list and the safe event list before beginning to simulate. We will present these tasks in the following.

The function schedule(), deque(), cancel() of the scheduler are modified for ELP approach because all of them access the event list or the safe event list. As Fig. 4.1 depicts, there is a list in the architecture of the NS2 network simulator and its scheduler is responsible for maintaining this list which stores unprocessed events. The scheduler provides three API function to control it – schedule(), deque(), and cancel(). We briefly describe the purpose of these functions.

- **schedule()** Insert a new event, which is provided by the caller, into the event list.

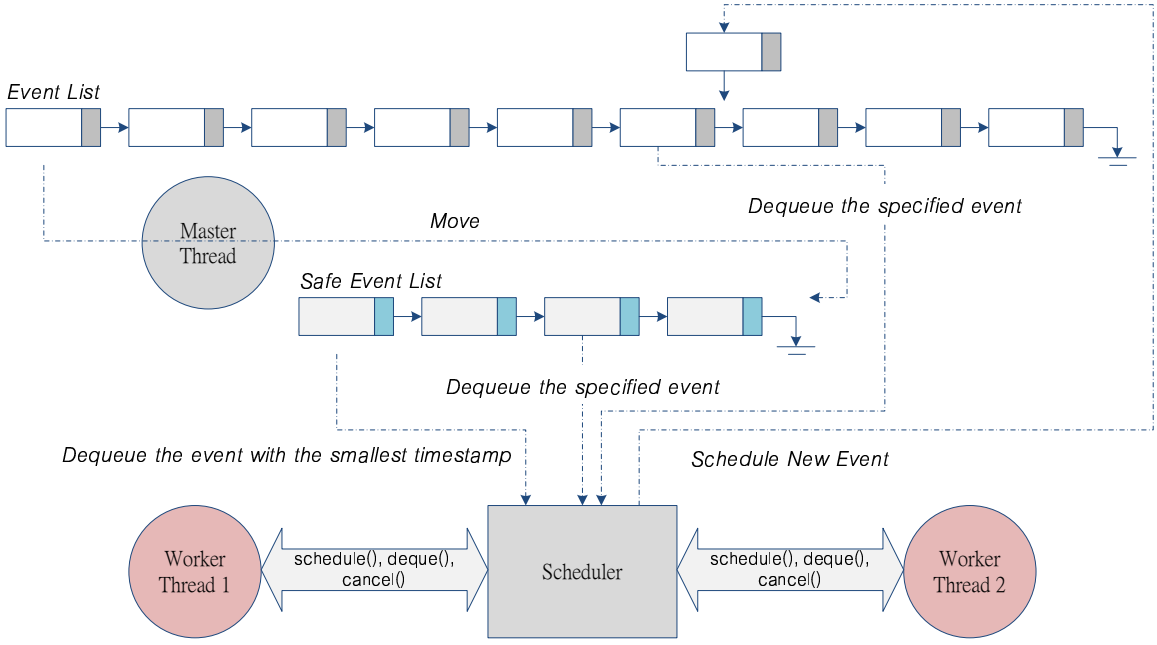


Figure 4.2: The relationship between the event list, the safe event list, and the scheduler of the NS2 network simulator using the ELP approach for dual-core system

- **deque()** Dequeue the event with the smallest timestamp from the event list.
- **cancel()** Remove the specified event, which is provided by the caller, from the event list.

Fig. 4.2 shows that there are two lists, which are the event list and the safe event list, in the network simulator using the ELP architecture and we need to modify the above functions of the scheduler to apply the ELP approach. The simple way to modify the function deque() is that we regard the original list as the safe event list. This way we do not need to change it because the worker thread requires a safe event with the smallest timestamp to execute by calling this function. The modified function schedule() inserts a new event into the event list instead of the safe event list (the original list). Nevertheless, the function cancel() needs to be extensively modified. This is because if the specified event, which the caller wants to remove, is not a safe event, then this function should remove it from the event list. On the other hand, if it has been a safe event, this function should remove it from the safe event list instead of the event list.

```
struct ELP_Info
{
      uint8_t flag_;
      unsigned int src_nid_;
      unsigned int dst_nid_;
      double time_;
      Event *owner;
      ...
};

class Event
{
   public:
         ...
         double time_;
         struct ELP_Info elp_info_;
         ...
}
```

Figure 4.3: The event data structure modified to support ELP approach

Both the event list and the safe event list need to be protected by locks when the function schedule(), deque(), and cancel() are accessing those; otherwise, some memory pointer errors will occur when there is more than one worker thread wanting to call these functions. Therefore, we insert some code for locking them using the POSIX Threads library to perform this.

4.1.2 Event Lists

Explained in the previous section, the original list in the NS2 network simulator is regarded as the safe event list in the ELP architecture. We need to create the event list to store unprocessed events which need not be determined whether they are safe events and a simple linked-list structure is used to implement it. In addition to the event list, the data structure of the "event" also needs to be modified. As Fig. 4.3 illustrats, we expand this data structure to store the source node ID and destination node ID of the event in the "src_nid_" and "dst_nid_" fields, respectively. That information will be used to determine whether this event is a safe event. To avoid null values in these fields, we assign the default value (65535) to it when an event is allocated and the event with the source node ID, 65535, is always determined as an unsafe event by the master thread.

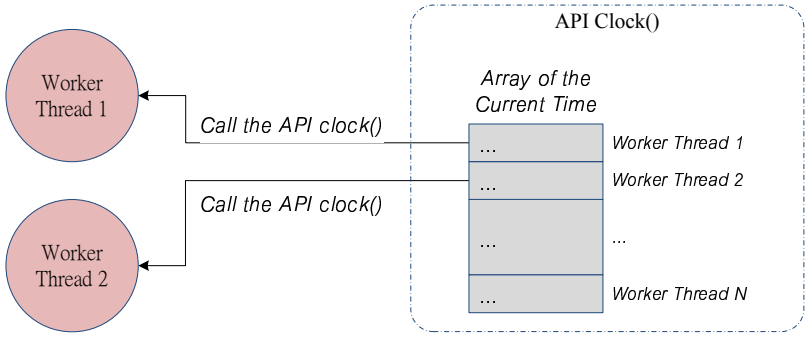


Figure 4.4: The worker thread gets its current simulation time based on its worker thread ID

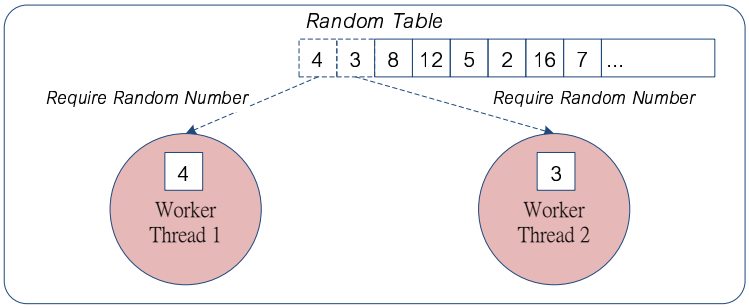
4.1.3 Simulation Clock

In the sequential simulation mode, there is only one executing event at any given time. As such, we need only one memory to store the current simulation time, which is accessed by an API function named "clock()." For the ELP approach, if there are N worker threads in the simulator, they may need to get the current simulation time during their event execution. However, the current simulation time of each worker thread may be different because it depends on the executing event in the worker thread.

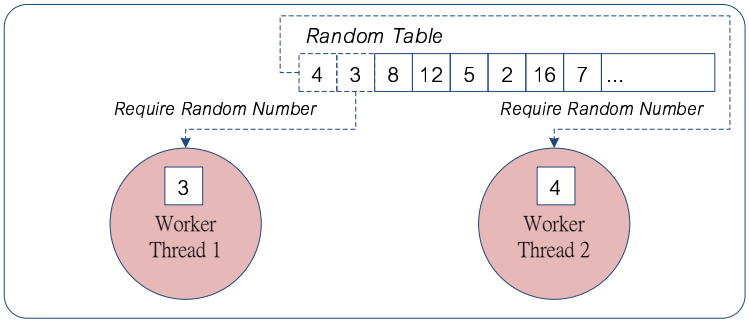
We must store these current time separately and modify the API function clock(), which is called by the worker thread to get the current simulation time. For this reason, an array variables is required to store these information and it is accessed based on the worker thread ID. As Fig. 4.4 depicts, when the worker thread executes an event and needs to get the current simulation time by calling the API function clock(), this API function retrieves the value from the array, which stores all current simulation time of the worker threads, based on the worker thread ID and return it as the current simulation time for this specific event.

4.1.4 Random Number

In the Linux system, a (single-threaded or multi-threaded) process is given a random table and it is shared among its user threads and kernel threads. However, this



(a) The worker thread 1 obtains a random number before the worker thread 2 obtains one.



(b) The worker thread 1 obtains a random number after the worker thread 2 obtains one.

Figure 4.5: A typical race condition for acquiring random number

situation leads to incorrect simulation results because there is a race condition among these worker threads. The problem of the race condition in the ELP approach is explained below using Fig. 4.5. In this figure, there are two worker threads in the network simulator and the random numbers, starting from the head, 4, 3, 8, 12, 5, ..., respectively, are in the random table which is used corresponded with this network simulator.

Suppose that worker thread 1 and worker thread 2 are respectively executing an event at the same time and they also want to obtain a random number during event execution. For example, worker thread 1 is executing the event $E1$ when performing the random back-off procedure, whose source node ID is N_1. It requires a random number to select one timeslot which is used to send a control or data packet. Worker thread 2 is executing the event $E2$, which is similar to $E1$, and the source node ID of $E2$ is N_2. As Fig. 4.5(a) illustrats, worker thread 1 and worker thread 2 both obtain

a random number. When worker thread 1 obtains a random number before worker thread 2 obtains one, their random numbers are 4, 3, respectively. On the other hand, if worker thread 1 requires a random number after worker thread 2 obtains one, then their random numbers are 3, 4, respectively, as Fig. 4.5(b) shown.

In the above example, when the worker threads get the different random numbers, the timeslot which are selected is also different. However, the order cannot be guaranteed because each of the event's executing time cannot be evaluated in any worker thread. In addition, the scheduler of the Linux system is one of the factors determining the order because the scheduler of the Linux system always selects the highest priority task to run. When there is more than one task with the same priority, it selects the head of the task list to run and the order of the task list is based on the order of the insertion (scheduling).

To solve the above problem, we use a simple hash function to replace the API function which returns a random number. The event's timestamp (tick) and its source node ID are used to compute a fake random number by the following simple hash function:

$$Fake\ Random\ Number\ =\ \{(Tick + SrcNID) \times 0x9e370001\} \bmod 2^{32}$$

Using the above hash function, the fake random number does not depend on the order of the worker thread, but it is still not the best solution.

4.1.5 Global Data Protection

There are two critical sections which need to be protected by locks in the NS2 network simulator applying the ELP approach. One is the recycled packet list and another is the log file description. The recycled packet list stores the events which are abandoned by simulation engine or other protocol modules. It means that the events need not be used and the NS2 network simulator collects those abandoned event in the recycled packet list waiting for next request. If this list is empty and someone requests an available event, the NS2 network simulator will dynamically allocate a memory space as an event and return it. When more than one worker thread requests an available

event at the same time, they may also access the recycled packet list. Therefore, this list needs to be protected by locks to avoid unexpected memory errors.

In addition, the log file description is a system file description which is used to access a file in the storage, such as a hard disk, a USB disk, a floppy, etc. It needs to be protected by locks because the NS2 network simulator stores all logs in the same log file. When more than one worker thread wants to write a record into this log file simultaneously, it becomes a critical section and needs to be protected by locks. Here, we can use the mutex lock of the POSIX Thread library to protect these critical sections.

4.2 The Components of the ELP Architecture

In addition to the modification to the NS2 simulation engine, there are new components of the ELP architecture that need be implemented. In this section, we present the design and implementation of the master thread, the worker thread, and the thread IPC among those. Finally, the design and implementation of the path lookahead are also presented.

4.2.1 The Master Thread

The master thread is responsible for finding the safe event set and moving them from the event list to the safe event list. As Fig. 4.6 illustrats, the function run() of the scheduler precomputes the minimum path lookaheads and creates the worker threads, the original execution flow will begin the main loop of the master thread to find the safe event set. The master thread repeats to find the safe event set based on the checking mechanism which are presented in Section 3.2. When the master thread has slept and waits for any worker thread to wake it up, it cannot wake up itself because it is a passive role in the ELP architecture.

In the procedure of finding safe events, we set the MP value to a value, which is the number of worker threads times 10. In the dual-core system, this value is 20. After

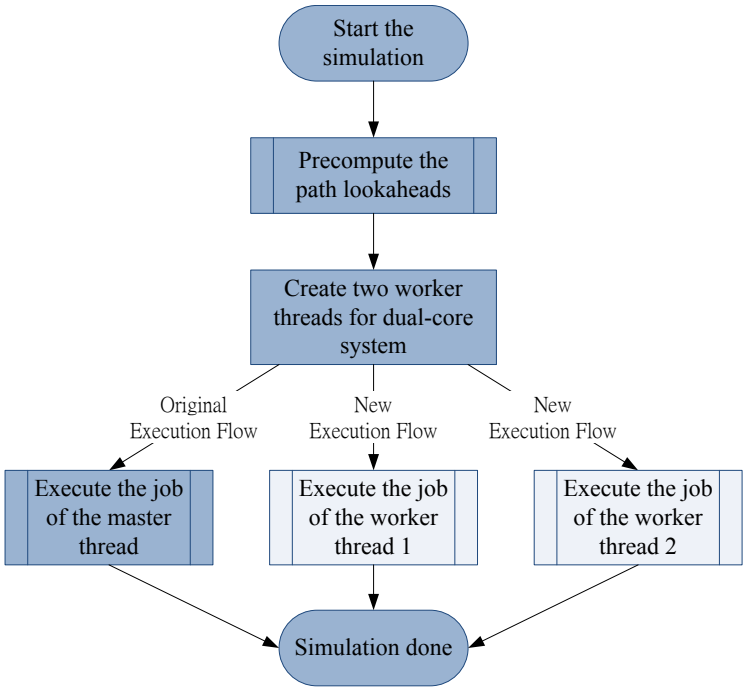
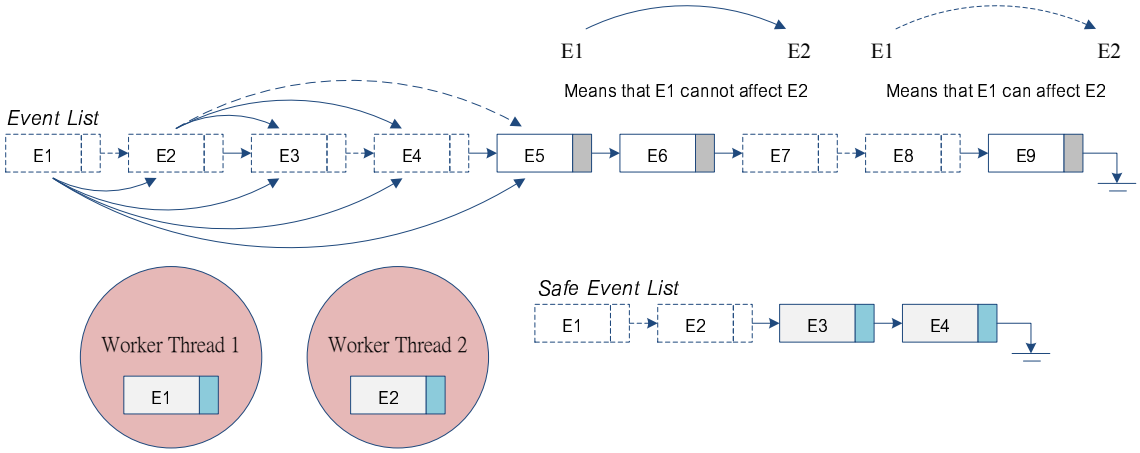


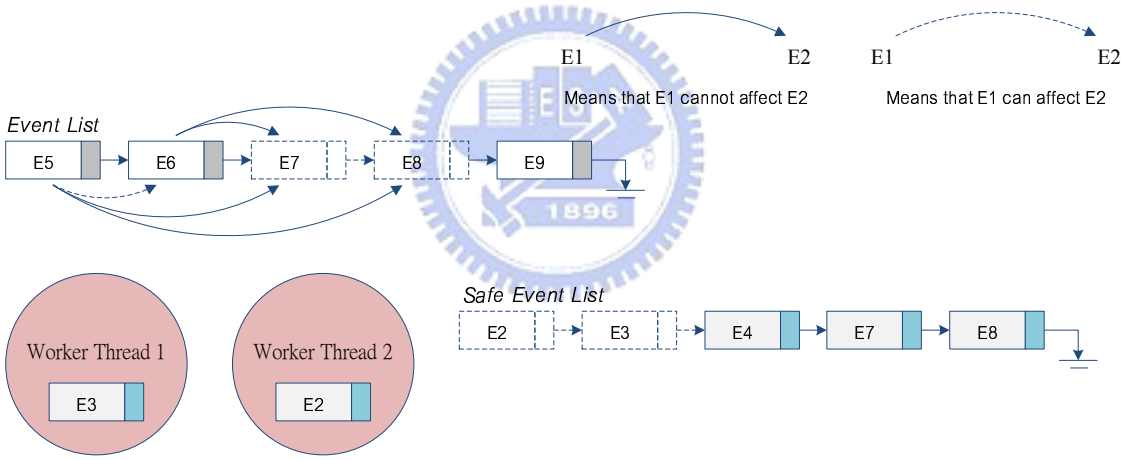
Figure 4.6: The flowchart for the modified run() function of the scheduler

the master thread sequentially finds MP safe events and moves them from the event list to the safe event list, it will stop to find safe event and enter idle mode waiting for any worker thread to wake it up again. On the other hand, if it cannot sequentially find MP safe events, the master thread will try to find at least two safe events for executing by the worker threads on the dual-core system. However, in order to find at least two safe events, the master thread may need to sequentially check far more than two events (e.g., 30 times checking) or it cannot found those in this situation. Therefore, we design a counter to store how many times the master thread checks. When this counter is greater than a threshold value, it will stop finding safe events. In our design, this threshold value is the same with the MP value. The above design can reduce the cost of the safe-event computation because this operation requires $O(MP^2)$ time complexity.

We give an example in Fig. 4.7 and the MP value is set to 20. Initially, suppose that there are nine events in the event list and the safe event list is empty. As shown in this figure, for discussion purposes we name the events stored in the event list,



(a) The master thread first enter the procedure of the finding safe event



(b) The master thread enter the procedure of the finding safe event again

Figure 4.7: The snapshots of the event list, the safe event list, and the two worker threads after the master thread performs two times finding-safe-event procedure

starting from the head, $E1$, $E2$, $E3$, $E4$, ..., respectively. If there is a solid arrow from event Ei to event Ej, this means that the master thread has determined that Ei cannot affect to Ej. On the other hand, if there is a dash arrow from event Ei to event Ej, this means that Ei can affect Ej. When the master thread first enter the finding-safe-event procedure, it is limited to move four events, starting from the head, $E1$, $E2$, $E3$, and $E4$ as the shown in Fig. 4.7(a). The reason is that $E2$ can affect $E5$ so that the master thread cannot sequentially find MP safe events. However, it has found four safe events (more than the number of worker threads), then it moves them from the event list to the safe event list and stops to continue finding safe event set. In Fig. 4.7(a), we use dash lines to draw $E1$, $E2$, $E3$, and $E4$ in the event list to represent the fact that they have been moved to the safe event list. We use dash lines to draw $E1$ and $E2$ in the safe event list to represent the fact that they have been dequeued by some worker threads and are still being processed.

When the master thread is waked up by any worker thread again, it enters the finding-safe-event procedure again. As shown in Fig. 4.7(a) and Fig. 4.7(b), we see that $E2$ can affect $E5$ and $E2$ is still processed in worker thread 1. Therefore, the master thread cannot determine whether $E5$ is a safe event and $E5$ can also affect $E6$. Suppose $E2$, $E3$, and $E4$ cannot affect $E7$ and $E8$, the master thread can move them from the event list to the safe event list and stop this finding-safe-event computation. This is because $E5$ and $E6$ are unsafe events. In this situation, the master thread only finds the first two safe events and it can provide adequate safe events for two worker threads in dual-core system.

4.2.2 The Worker Thread

The worker thread plays an active role in the ELP architecture. When the worker thread is created, it will try to dequeue a safe event from the safe event list to execute it. If there is no safe event in the safe event list, the worker thread will enter the idle mode waiting for the master thread to wake it up and it also ask the master thread to find more safe events for it. It can wake up itself by a sleep timeout, to avoid idle too

long. As Fig. 3.3 depicts, when the worker thread wakes up itself, there may be no safe event in the safe event list. In this situation, the worker thread will enter the idle mode again and its sleep time becomes doubled. If the worker thread can dequeue a safe event to execute, its sleep time will be reset to the default value (10us in our design).

In the guideline for using POSIX Thread library, it suggested that the user threads, which are created by the original execution flow (It is called "parent thread". In the ELP architecture, it is the master thread.), should not be terminated by the parent thread. It should terminate by itself because when the user thread is terminated by the parent thread when the user thread is executing some computations, it leads to unexpected errors. In our design, we use a flag, which is implemented by a global variable, to denote whether the simulation has done. When the worker thread has finishes its current event computation, it will check this flag to determine whether it should terminate itself. This global variable is controlled by the master thread. When the master thread has set the flag to on, it will wait for all worker threads to terminate. Then it terminates the whole simulator.

However, if the worker thread asks the master thread to find more safe events each time when it finds the safe event list empty. The worker thread will enter the idle mode very often and waste some CPU cycles. Therefore, we propose a mechanism to solve this problem. When the worker thread has dequeued a safe event from the safe event list, it will check the length of the safe event list to see whether the length is under a threshold value, which is a low bound for the length of the safe event list. In our design, we set the threshold value to the number of worker threads plus one. For a dual-core system, this value is set to 3. Therefore, when the worker thread finds that the length of the safe event list is under this value, it will ask the master thread to find more safe events early and continue to execute its current event computation to avoid wasting CPU cycles.

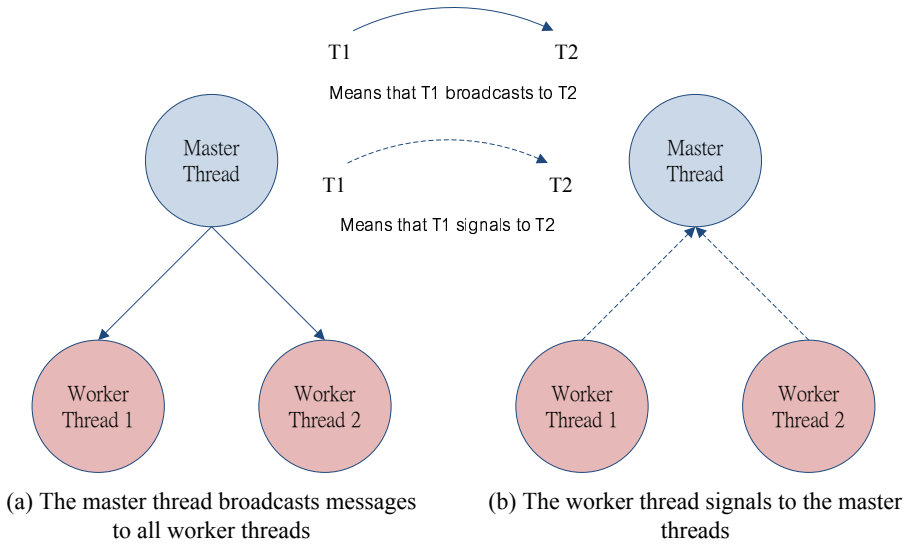


Figure 4.8: There are two different control messages between the master thread and the worker threads

4.2.3 The Thread IPC

As aformentioned, there are many control messages exchanged between the master thread and worker threads. We can divide these control messages into two different directions as shown in Fig. 4.8. One is the master thread signals the worker threads for continuing to dequeue a safe event and execute it. The other is the worker threads ask the master thread to find more safe events for them.

In the POSIX Thread library, there are two API function, pthread_cond_signal() and pthread_cond_broadcast() that can be used to send messages from one user thread to other user threads. The function pthread_cond_signal() can signal to a specific thread and the function pthread_cond_broadcast() can broadcast messages to a specific group of threads.

When the master thread wants to signal the worker threads for continuing to execute safe events in the safe event list, it uses the function pthread_cond_broadcast() to broadcast the message to all worker threads. However, if we always adopt this approach, the worker threads will be waked up by the master thread every time and checks the safe event list whether there is any safe events need to be executed. If the

number of the safe events in the safe event lists is not adequate for all worker threads, some of them will be waked up by the master thread but there is no safe event can be executed by them.

In this situation, these worker threads have no safe event to execute and they will waste some CPU cycles and increase the overhead of locking the safe event list. Therefore, when the number of the safe event list is not adequate for all worker threads, the master thread uses the function pthread_cond_signal() instead of pthread_cond_broadcast() to wake up a certain number of worker threads, but not all. For example, if there is only one safe event in the safe event list, the master thread signals to only one of the worker threads to execute it using the function pthread_cond_signal().

On the other hand, when the worker threads wants to ask the master thread to find more safe events, they always use the function pthread_cond_signal() to wake up the master thread because there is always a master thread in a simulator.

4.2.4 The Path Lookahead

For the ELP approach, the path lookahead directly affects how many safe events can be found by the master thread. As mentioned in Section 3.2.1, we need to compute the minimum path lookaheads between all pairs of nodes. When computing all pairs shortest paths, we use the Floyd-Warshall algorithm to precompute and store the minimum path lookaheads for all pairs of nodes in a two-dimension array (it is called "path lookahead table"). The path lookahead table requires $O(N^2)$ memory space to store these result. The master thread can easily access this result based on the source and destination node ID of a shortest path.

However, before computing all-pairs shortest paths, we need to create the initial matrix, which is used by the Floyd-Warshall algorithm, to store the link delay between any two nodes in the simulated case. A simple way to implement this is using an NxN array, which is similar to path lookahead table and "N" is the total number of nodes, to store them.

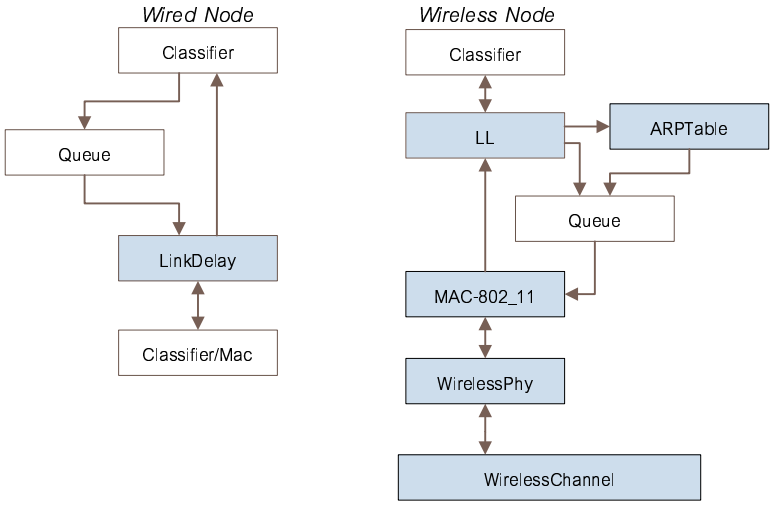


Figure 4.9: The wired and wireless network protocol modules

In the wired network of the NS2 network simulator, there is a link associated with a link delay between two nodes. When the simulator is reading the simulated case, it will synchronously fill the link delay in the initial matrix for computing the minimum path lookaheads. Nevertheless, the NS2 network simulator does not use the same design in its wireless network. In the wireless network, the link delay between any two nodes is computed based on its location. When the master thread precomputes the minimum path lookaheads, it needs to fetch the location information of each node for that.

As mentioned in Section 3.2, the lookahead value of packet arrival events is the sum of the propagation delay and the transmission time. However, the transmission time depends on the packet size so that it cannot be easily computed because it is a variable. In our design, the lookahead value of a packet arrival event equals the link delay.

4.3 Modifications the NS2 Network Protocol Modules

In this thesis, we focus on the network protocol modules of the wired network and the wireless network built on top of the NS2 network simulator. The wired and wireless network protocol modules diagram are shown in Fig. 4.9. Protocol modules colored with light blue (or light gray) are modified by us to support the ELP approach. In the following section, we briefly describe some modification of these.

4.3.1 Wired Protocol Modules

We use a simple wired network to simulate our case. In the simple wired network, the NS2 network simulator focuses on the behaviors of the interface queue and the link delay to simulate. Only the module LinkDelay needs to be modified to support the ELP approach in this mode. We briefly describe Queue and LinkDelay in the following.

- **Queue**

  This module simulates the behaviors of the interface output queue and is responsible for queuing the packets which wait to be sent on the wired.

- **LinkDelay**

  This module performs the important functions including the link delay and scheduling a packet arrival event. It will schedule a packet arrival event to simulate a packet sent on the wired and schedule a local computation event to poll the next packet waiting to be sent in the interface queue. Before these events are scheduled, it needs to tag these events with its source and destination node ID and store them in the fields of the event data structure.

4.3.2 Wireless Protocol Modules

There are six protocol modules in the wireless network of the NS2 network simulator. We modify these modules, LL, ARPTable, MAC-802_11, WirelessPhy, and WirelessChannel, to support the ELP approach. We briefly describe them in the following.

- **LL**

  This module is responsible for controlling the ARPTable module whether it needs to send ARP packet to look up a destination MAC address and schedules some local computation event. Similarly, it needs to tag these with its source and destination node ID and store them in the fields of the event data structure.

- **ARPTable**

  This module performs the mechanism of the ARP protocol and it also schedules some local computation events which need to be tagged with the source and destination node ID.

- **MAC-802_11**

  This module performs all functions of the 802.11b network protocol. Furthermore, there are many local timers to control the mechanism of the 802.11b network protocol. Each local timer also schedules a local computation event and they are tagged with the source and destination node ID.

- **WirelessPhy**

  This module simulates all behaviors of the wireless network and some local computation events are used to control it. WirelessPhy also tags these events with its source and destination node ID.

- **WirelessChannel**

  In the wireless network, this module is shared among all nodes because it performs the wireless channel simulation. When it receives a packet from the upper-layer which wants to send a packet, it will schedule a packet arrival event. They are tagged with the source and destination node ID.

4.4 Modificatios to the NS2 Traffic Generators

A traffic generator is responsible for periodically creating a packet and sending it to a specific node in a network topology. There is a local timer in a traffic generator, which is used to periodically perform that. In the NS2 network simulator, all the traffic generators inherit the C++ class "TrafficGenerator" and there is a function timeout() in this class to control its local timer. We can only modify this function to tag the local computation event with the source and destination node ID, which are used by its local timer.

4.5 An Advanced Mechanism

As mentioned in Chapter 3, we understand that the performance speedup depends on the degree of ELP. In order to increase the degree of ELP, we propose to apply the group safe event checking condition to the finding-safe-event procedure. However, if the computation cost of the finding safe event is greater than the one of processing safe event, the performance speedup may be smaller than 1. Therefore, we propose a mechanism, which is called "Automatic Mode Switching (AMS)", trying to solve this problem. In the follow section, we present these new mechanisms respectively.

4.5.1 The Group Safe Event Condition

In the wireless network, when a node i sends a packet the WirelessChannel module will schedule a packet arrival event from node i to each node within the interference range of node i. If the distance between one node and its neighboring nodes is the same, then these packet arrival events have the same source node ID and the schedule time (it means that the event will be executed at this time). Therefore, these packet arrival events are considered safe for each other.

The reasons for the above checking rules are explained below using Fig. 4.10. In this figure, each circle drawn using solid line denotes a wireless node in the network. A node has its name Ni, where i denotes the ID of the node. The bigger circle drawn

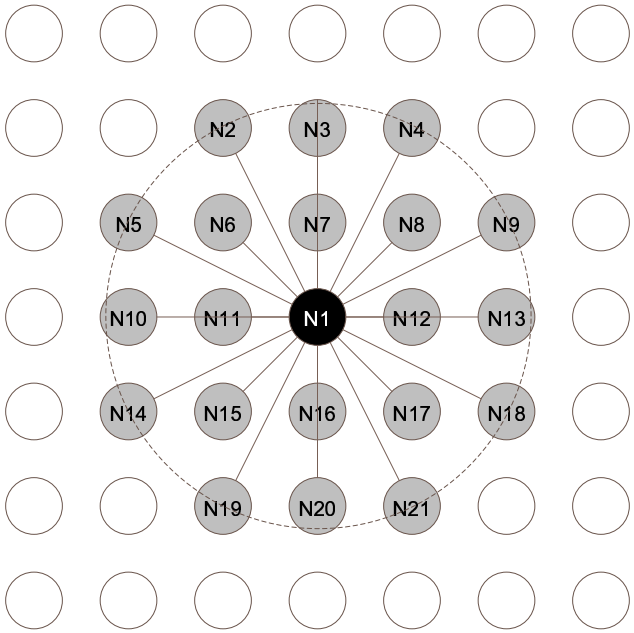


Figure 4.10: A 7x7 grid wireless network used to ilustrate group safe events found at the event level

using dash line represents the interference range of the node $N1$ colored with black. When node $N1$ wants to send a packet, it needs to schedule a packet arrival event to each node within the interference of node $N1$. As Fig. 4.10 shows, there are twenty nodes within the interference of the source node $N1$. One can view that there is a link between the source node $N1$ and these destination nodes, which are named $N2$, $N3$, $N4$, $N5$, ..., respectively.

Suppose that the distance from one node to its neighboring nodes is the same. These destination nodes can be divided into four groups. One includes $N3$, $N10$, $N13$, and $N20$. Another includes $N7$, $N11$, $N12$, and $N16$. Another includes $N6$, $N8$, $N15$, and $N17$. The other includes $N2$, $N4$, $N5$, $N9$, $N14$, $N18$, $N19$, and $N21$. They have the same source node ID and the events of each group have the same scheduled time. In this example, suppose that a packet arrival event $E2$ from $N1$ to $N2$ and another packet arrival event $E4$ from $N1$ to $N4$ are scheduled. There is a minimum path lookahead PLA_{24} of the path from $N2$ to $N4$ and the timestamp of $E1$ is equal to that of $E2$. The timestamp of $E1$ plus PLA_{24} is greater than that of

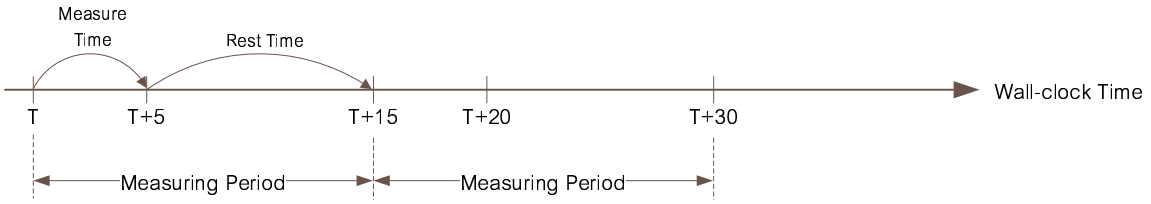


Figure 4.11: The measuring period of the AMS mechanism

$E2$. Therefore, $E1$ cannot affect $E2$. Similarly, $E2$ cannot affect $E1$, either.

For the above result, when the packet arrival events can be classified to the same group. They cannot affect each other.

4.5.2 Automatic Mode Switching

Using the ELP approach, there are additional computations that need to be performed in a network simulator because the network simulator is responsible for determining the safe event set and moving them from the event list to the safe event list. In addition to that, there are more locking overhead when the worker threads want to access a global data structure simultaneously. There are the ELP overhead for a network simulator. If the ratio (we call it ELP overhead ratio) of additional computations to the ones of processing event is too high, the performance speedups may be smaller than 1 and can be as low as 0.5. If the ELP overhead ratio is greater than a value, which is a system parameter, the network simulator will turn on the AMS mechanism. When it is turned on, the master thread will switch the operation mode of the network simulator and terminate all worker threads. In this situation, the operation mode of the network simulator is restored to the sequential one, which is similar to its sequential version.

As Fig. 4.11 shows, suppose that the current wall-clock time is T, we measure the average computing time C_o for finding safe event and the average processing time C_p for executing a safe event duration a 5-second. The wall-clock time $T + 5$ is a checkpoint where one needs to check whether the ratio of C_o to C_p is greater than 1. If the ratio is greater than 1, it means that there are too many effort spent on finding

safe event. When the network simulator detects this situation at this checkpoint, it will switch the operation mode of the simulation to the sequential one. On the other hand, if the ratio is smaller than 1, the network simulator keeps the multi-threaded mode to continuously simulate and waits 10 seconds for measuring again. The above mechanism is enabled during the simulation execution until the simulation is done or the operating mode has been switched to the sequential mode.

Chapter 5

Performance Evaluation

We first explored the degree of ELP that can be found in network simulations under various network conditions. This feasibility study is important for the ELP approach because if more ELP can be found, the potential performance speedups provided by the ELP approach will be higher. We also performed sequential and parallel network simulations using the ELP approach and measured the performance speedups under various network conditions on dual-core systems.

The overhead of the ELP approach includes the computation time for finding safe event set, locking for global data protection, and the thread IPC. These are important factors for the performance speedups. We only measure the time for finding the safe event set as the ELP overhead. Here we define the performance speedups and the ELP overhead in percentage as follows.

$$Performance\ Speedup = \frac{Sequential\ Execution\ Time}{Parallel\ Execution\ Time}$$

$$Percentage\ of\ ELP\ Overhead = \frac{Master\ Thread\ Execution\ Time}{Whole\ Execution\ Time}$$

We used the program "top" to obtain execution time of each thread and the program "time" of the GNU project to obtain the CPU utilization. We used 10 dual-core systems (they are of the same brand name and model) for these experiments. Each of them has an Intel Core 2 CPU 6400 running at 1.86 GHz and has 1GB main memory. The operating system running on these systems is Fedora Core 6 with the

version-2.6.20.4 of the Linux kernel. During experiments, the network interfaces on these systems were all shut down to avoid external interferences from the network. To avoid internal interferences from the various service daemons running in the background of the operating system, all unnecessary services were disabled. In addition, since the CPUs can dynamically change their clock rates for power management, we fixed the clock rates of the CPUs so that performance speedup measurements can be correctly performed. In the ELP approach, the POSIX Thread library (it is also called PThread library) implementing the 1x1 model of NPTL is used to implement the parallel network simulator. In Linux, user threads created by PThread library are associated with Linux kernel threads.

To fairly compare the simulation speeds of a sequential network simulation and its corresponding parallel network simulation, every sequential network simulation was run on top of the SMP-version Linux kernel. We found that this setting is very important and can affect the measured performance speedups greatly. If the SMP mode is disabled when the sequential network simulation is run, the simulation will take more time to finish. As such, a higher performance speedup will result. This is because although only one CPU can be used for the execution of the sequential network simulation, the other CPU can be used to service system daemons that cannot be shut down for their importance. If instead the sequential network simulation is run with the SMP mode disabled, only one CPU can be used and it needs to run the sequential network simulation while servicing system daemons. In such situation, a performance speedup higher than 2 may be measured on a dual-core system, which clearly is against the intuition.

In the following section, we report the ELP degree, performance speedups and the ELP overhead on wired and wireless network.

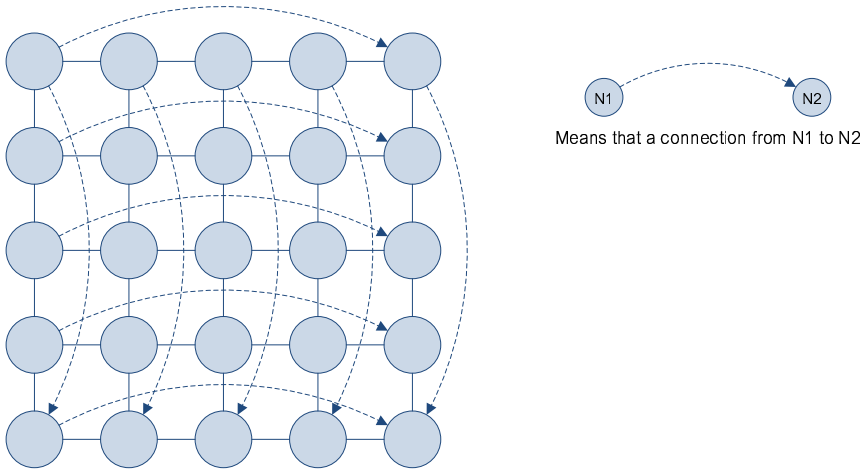


Figure 5.1: The connection pattern on a 5x5 grid wired network

5.1 Traffic on Wired Networks

5.1.1 System Paramemters

The common simulation parameters listed in Table 5.1 are used for wired networks. In this table, the bold and italic strings are the default value when that system parameter is not varied. We describe these system parameters in detail as follows.

The topology of the evaluated wired network is a NxN grid network, where N is a system parameter called "Network topology size" and is varied from 5, 6, 7, 8, 9, to 10. When this parameter is not varied, the default value for it is 10. Each node in the grid has a link connecting itself to each of its neighboring nodes in the pattern shown in Fig. 5.1.

Table 5.1: Common simulation parameters for wired networks

Parameter name	**Value**
Network topology size	5, 6, 7, 8, 9, to ***10***
Link delay	***5ms***, 10ms, 20ms, to 30ms
Link bandwidth	10Mb, ***100Mb***, to 1000Mb
Coding computation loop	***0***, 128K, 256K, 512K, 1024K, 2048K, to 4096K

The delays and bandwidths of all links are the same. They are system parameters called "Link delay" and "Link bandwidth", respectively. The link delay is varied from 5ms, 10ms, 20ms, to 30ms. When this parameter is not varied, the default value for it is 5ms. The link bandwidth is varied from 10Mb, 100Mb, to 1000Mb. When this parameter is not varied, the default value for it is 100Mb.

The packet size which is viewed at the transport layer is always 1000Bytes, despite whether it is for UDP connections or TCP connections. Furthermore, the packet transmission time on these links is a varied value based on the link bandwidth because the packet transmission time is the packet size divided by the link bandwidth and the packet size is a fixed value.

The last system parameter, called "Coding computation loop," represents an iteration counter value. We used a "for" loop with this iteration counter value to simulate the computation time of channel coding before scheduling the packet arrival event at the physical layer. It is varied from 0, 128000, 256000, 512000, 1024000, 2048000, to 4096000, representing different computation time of channel coding used to perform the computation. The total number of times for performing channel coding equals the total number of the packet arrival events. When this parameter is not varied, the default value for it is 0. This means that no additional computation needs to be performed.

We created UDP and TCP connections for each case and respectively measured the ELP degree and performance speedups. In order to scatter all events over each node, we created the UDP or TCP connections from nodes located at the top and left side of the grid network, to the opposite side nodes. An example for the connections pattern of a 5x5 grid network is illustrated in Fig. 5.1. The traffic generator for UDP connections periodically sends packets at a constant bit rate and the interval time of it is 50us.

Besides the cases with varied coding computation loop, the simulated time for each case is 30 seconds despite whether it is for ELP degree evaluation or for performance speedups measurement. The simulated time for the cases with the coding computation loop is set to 2 seconds because they need more time to perform simulation. When

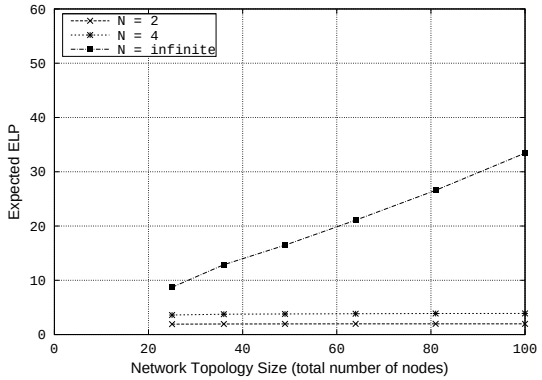
all system parameters of one case is not varied, we call it "Standard case."

On these dual-core systems, each simulation case usually took a time between 1 and 24 hours to finish, depending on the total number of packet events that need to be processed in a simulation. It is important to make sure that the ELP approach is feasible and generates correct simulation results. As such, for each case, we compared the simulation results (including the detailed packet log) generated by the sequential simulation approach against those generated by the ELP approach. We found that, the ELP approach generates exactly the same simulation results as those generated by the sequential simulation approach.

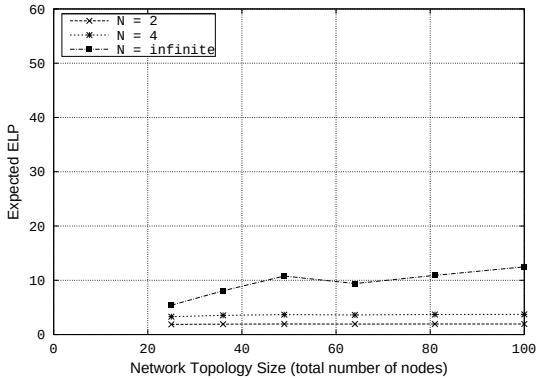
5.1.2 ELP Degree and Performance Speedups

To explore the maximum ELP available in network simulations, we modified the worker thread and master thread designs in the following way. Every time when a worker thread tries to dequeue a safe event from the safe event list, it first wakes up the master thread using the function pthread_cond_signal(). For the master thread, every time when it is waked up, it finds the largest safe event set (by using a very large value, say 1000, for the MP system parameter discussed in Section 3.3), moves the events in this set to the safe event list, and then sleeps. Concurrently, the worker thread logs the current number of safe events in the safe event list (including the event that may be executed by the other worker thread). In our study, a simulation case usually generates more than ten million such samples, and these samples are used to compute the expected ELP of a simulation.

When using these samples to compute the expected ELP of a simulation, we compute three different versions of the expected ELP. The first version is computed based on the assumption that one has an infinite number of CPUs. That is, one has a N-core system, where N is ∞. The first version represents the maximum expected ELP available in a network simulation. The second version is computed based on the assumption that N is 4 (quad-core) and the third version is computed based on the assumption that N is 2 (dual-core). When computing the second version, every

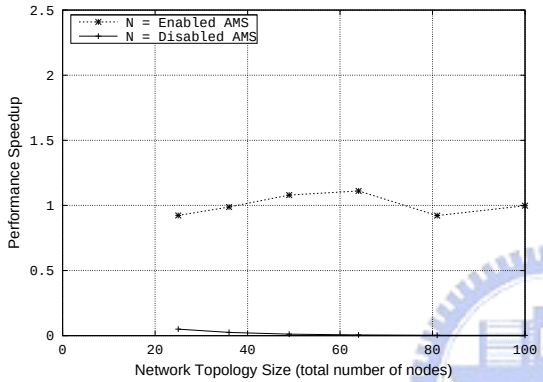


(a) UDP connections

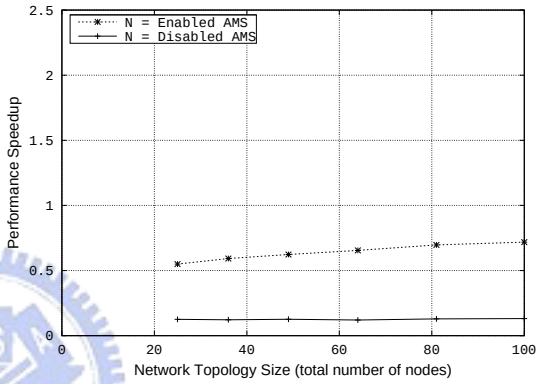


(b) TCP connections

Figure 5.2: Expected ELPs on a wired network: The network topology size is varied.



(a) UDP connections



(b) TCP connections

Figure 5.3: Performance speedup on a wired network: The network topology size is varied.

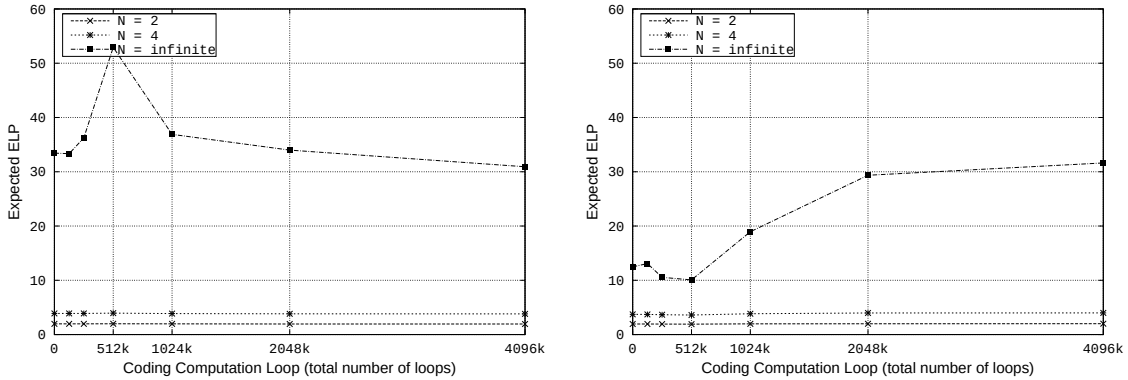
sample whose value greater than 4 is reduced to only 4 to account for the fact that at any time at most four CPUs can be active executing events. As such, the second version of the expected ELP will always be less than 4. The processing for computing the third version is similar. Any sample whose value is greater than 2 is reduced to only 2, and as such the third version of the expected ELP will always be less than 2. We also measure two different versions of the performance speedup. The first version is the performance speedup with the ASM mechanism enabled. The second version is the performance speedup with the ASM mechanism disabled.

From Fig. 5.2(a) and Fig. 5.2(b), one sees that when a simulated network has

more nodes, its expected ELPs increase. This property can be explained with the safe event checking rules used in the ELP approach. In Table 3.1, as long as $DstNID_1 \neq DstNID_2$, although a path lookahead condition still needs to be met, it is very likely that event $E1$ cannot affect event $E2$ and thus they can be executed in parallel. Therefore, if the number of nodes in a network increases, it becomes more and more likely that the condition $DstNID_1 \neq DstNID_2$ can hold. This property is good for large-scale wired networks, ad hoc networks, and sensor networks, where hundreds or thousands of nodes exist in the network. These large-scale networks also are the ones that demand high performance speedups the most. For a network with only a few nodes (e.g., 4), the ELP approach may not generate good performance speedups because the chances that the condition $DstNID_1 \neq DstNID_2$ holds are small.

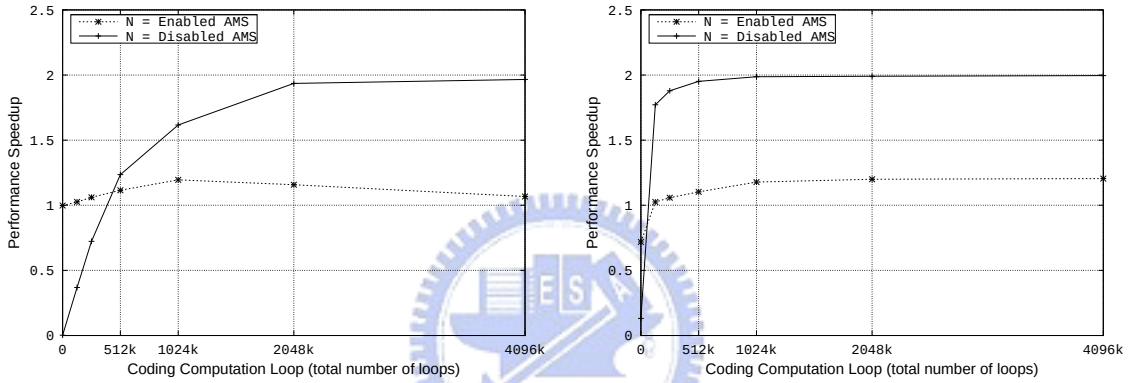
From these plots, one sees that the expected ELPs of UDP connections are greater than that of TCP connections. This is because the window size of TCP protocol is set to 20Bytes by default in the NS2 network simulator and the maximum throughput of TCP connections is 500Kbytes/sec. However, the interval time of UDP traffic generators is 50us and the packet size is fixed to 1000Bytes. In such parameters, the UDP traffic generator can result 160Mbytes/sec, but the maximum bandwidth of all links is 100Mbytes/sec in these case. The throughput of UDP connections are far greater than that of TCP connections. Therefore, the total number of events of UDP connections is more than that of TCP connections. For the above reason, the expected ELPs of UDP connections are greater than that of TCP connections.

From Fig. 5.3(a), one sees that on a dual core system, the performance speedups of UDP connections on wired network are about 1 under different network topology sizes with the AMS mechanism enabled. When the AMS is disabled, the performance speedups are close to zero. Although there are abundant ELP in network simulation, the ELP approach could not generate good performance speedups. This is because the speed of dequeuing safe events by worker threads is far faster than the speed of enqueuing safe events by master thread because the average computation time of events is very small. Similar results for TCP connections are shown in Fig. 5.3(b). From most performance speedups plots, one sees that most plots have the similar



(a) UDP connections (b) TCP connections

Figure 5.4: Expected ELPs on a wired network: The coding computation loop is varied.

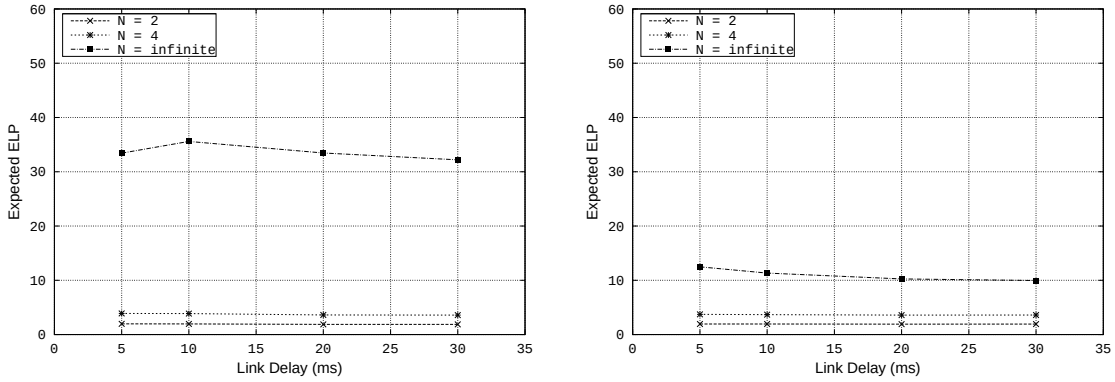


(a) UDP connections (b) TCP connections

Figure 5.5: Performance speedup on a wired network: The coding computation loop is varied.

results.

From Fig. 5.4(a) and Fig. 5.4(b), one sees that when the coding computation loop is between 0 and 512000, the expected ELPs increase. This is a natural result because when the worker threads need to spend more time on execution event, there are more safe events in the safe event list and the ELP degree increases. However, as shown in Fig. 5.4(a), the expected ELPs decrease when the coding computation loop is greater than 512000. This can be explained because when the coding computation loop is too large, the speed of scheduling other events during execution of this packet arrival



(a) UDP connections

(b) TCP connections

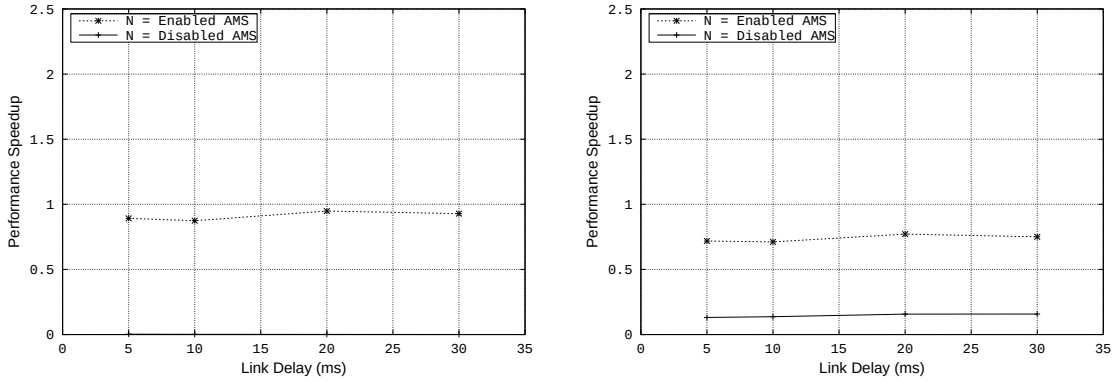
Figure 5.6: Expected ELPs on a wired network: The link delay is varied.

event will decrease. As a result, the master thread cannot find more safe events and move them from the event list to the safe event list. Although the expected ELPs decrease when the coding computation loop becomes large, there are abundant ELP in network simulations yielding good performance speedups result.

The above results depend on our design of the ELP-degree version of the NS2 network simulator. As aforementioned, every time when a worker thread tries to dequeue a safe event from the safe event list, it first wakes up the master thread using the function pthread_cond_signal() in this version. Furthermore, after signaling to the master thread, the worker thread will continue to execute the safe events. In this situation, the speed of enquening safe events by master thread, the one of dequeuing safe events by worker thread, and the one of scheduling other events during execution of a safe event can affect the expected ELPs. This is because the master thread and the worker threads may run simultaneously.

From Fig. 5.5(a) and Fig. 5.5(b), one sees that if there are more computations need to perform, the higher performance speedups on a dual-core system will be achieved. This results show that the previous treatments are correct. The speed of dequeuing safe events by worker threads far faster than the speed of enqueuing safe events by master thread cannot result good performance speedups.

From Fig. 5.6(a), Fig. 5.6(b), Fig. 5.7(a), and Fig. 5.7(b), one sees that the delay of the links in a network affects the expected ELPs and the achieved performance



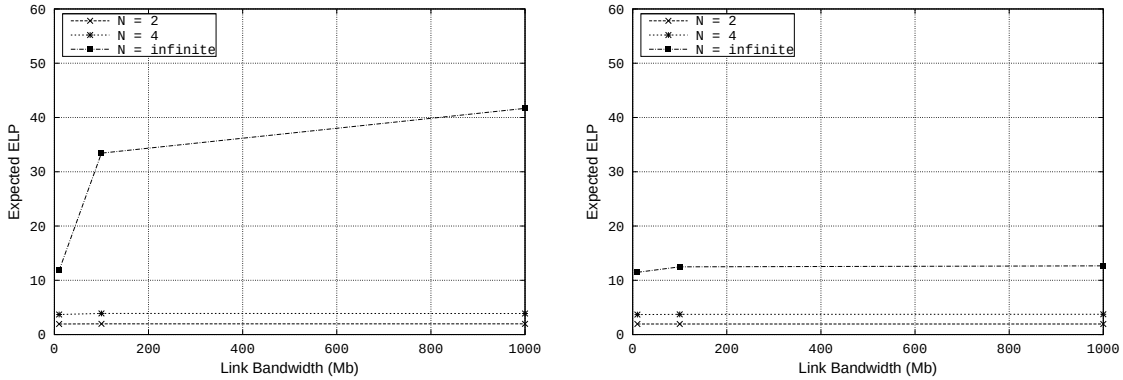
(a) UDP connections

(b) TCP connections

Figure 5.7: Performance speedup on a wired network: The link delay is varied.

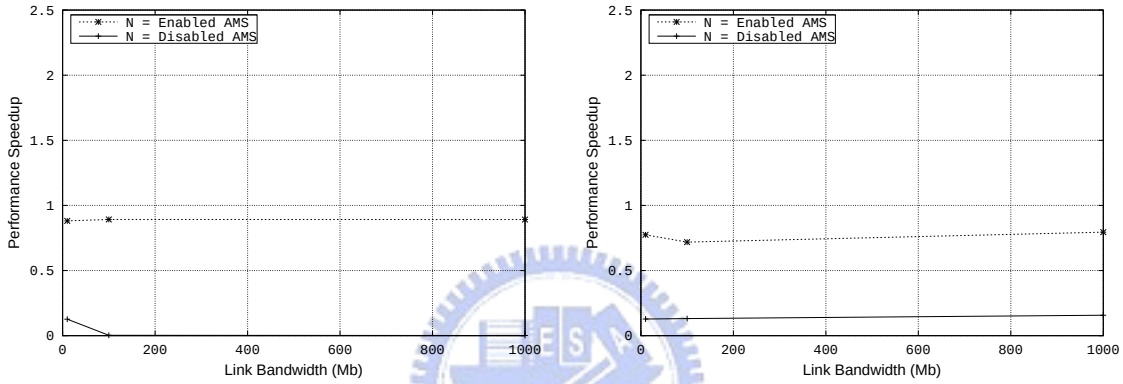
speedups minimally. One may explain that, like in the conservative approach, using a larger value for the link delay would increase link and path lookaheads and thus would increase parallel network simulation performance. However, the results show that this is not the case. Since a small link delay results in the same performance speedups as a large link delay does, the ELP approach does not suffer the tiny lookahead problem, which causes the conservative approach to perform very poorly. This property is good for network s in which most links are short links such as Ethernet links or IEEE 802.11 (a/b/g) wireless channels.

The reason for this property is explained below. Suppose that one increases the link delay by a factor of N, the time axis now is decreased by a factor of N. This is because when a packet is forwarded by nodes in the network during simulation, packet arrival events will be scheduled on the time axis at an interval of the link delay. The reason is that a packet must traverse a link to arrive at the next node. When the link delay is increased by a factor of N, the path lookahead computed between two nodes in the ELP approach will also increase by a factor of N. Although the increased path lookahead spans a larger interval on the time axis, the number of events "covered" (meaning that these events become safe events due to this path lookahead) by it remains about the same because the average interval between two neighboring events on the time axis is also increased by the same factor. As such, the effect of the increased path lookahead is canceled out by the effect of the decreased



(a) UDP connections (b) TCP connections

Figure 5.8: Expected ELPs on a wired network: The link bandwidth is varied.

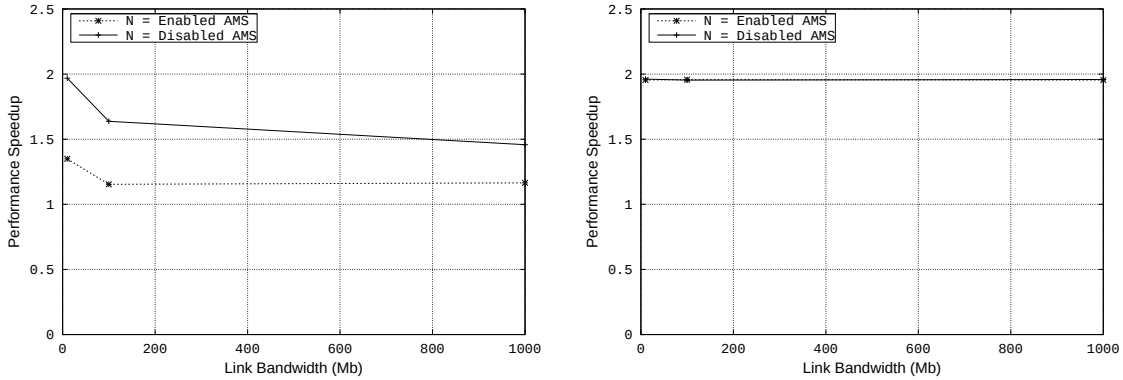


(a) UDP connections (b) TCP connections

Figure 5.9: Performance speedup on a wired network: The link bandwidth is varied.

packet density. This is why the link delay affects the expected ELPs and performance speedups minimally in the ELP approach.

From Fig. 5.8(a), one sees that the bandwidth of the links in a network affects the expected ELPs clealy. One may think that using a larger link bandwidth would increase the event density on the time axis and thus increase the expected ELP of a simulation case. In NS2 network simulator, the window size of TCP protocol is a fixed value by default and the event density on the time axis thus depends on the window size and the link delay. Therefore, the window size limits the event density on the time axis so that the expected ELPs are limited by it as shown in Fig. 5.8(b). The results of the performance speedups shown in Fig. 5.9(a) and Fig. 5.9(b) have the



(a) UDP connections

(b) TCP connections

Figure 5.10: Performance speedup on a wired network: The link bandwidth is varied and the coding computation loop is 2048000.

same situation for the case under different network topology size and can be explained using the same explanations.

In order th prove the previous conjecture, we create other simulated cases under different link bandwidth and the coding computation loop is fixed to 2048000. Other system parameters are not varied for this case and the simulated time for it is 5 seconds. The performance speedups of UDP or TCP connections on a wired network are also measured. The results of the performance speedups are shown in Fig. 5.10(a) and Fig. 5.10(b).

From Fig. 5.10(a) and Fig. 5.10(b), one sees that when there are additional computations need to performed good performance speedups will result. The link bandwidth in a network still affects the performance speedups of TCP connections minimally. However, when the link bandwidth increases, the performance speedups slightly decrease. One may think that, when the link bandwidth increasing the more work need to perform, there are more events in the event list because the event density on the time axis increases. As such, the master thread needs to perform many times checking for finding the safe event set. On the other hand, the higher ELP degree does not mean that the master thread can readily find the safe event set for the worker thread. This is because when the master thread cannot sequentially find MP events, it tries to only find at least two events for worker threads to execute it. The above reason

can explain the performance speedups as shown in Fig. 5.10(a), but the performance speedups still are more than 1.4 with the ASM mechanism disabled.

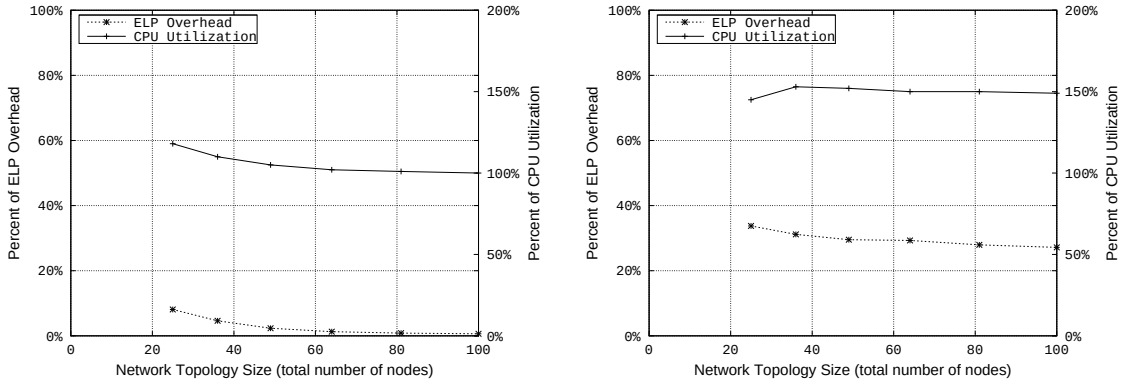
5.1.3 ELP Overhead in Percentage

We measured the ELP overhead and the CPU utilization in percentage under various system parameters. In Linux system, the max CPU utilization of each CPU is measured as 100% and the max CPUs utilization of the whole system is the sum of the utilization of each CPUs. Therefore, the max CPUs utilization of a dual-core system is 200%. Similarly, the max CPUs utilization of a quad-core system is 400%. We will use the "CPU utilization" to refer to "CPUs utilization of the whole system" in the rest of this thesis when there is no ambiguity.

From Fig. 5.11(a), Fig. 5.13(a), and Fig. 5.14(a), one sees that the CPU utilizations are close to 100% and the ELP overheads are very low (under below 20%). It means that there are abundant ELP in these simulated cases but the average computation time of each event is too small. In this situation, the master thread can readily find the safe event set and move them from the event list to the safe event list but the speed of event execution is too high. The worker threads are often idle waiting for safe events to execute. Therefore, the achieved performance speedups are not too high.

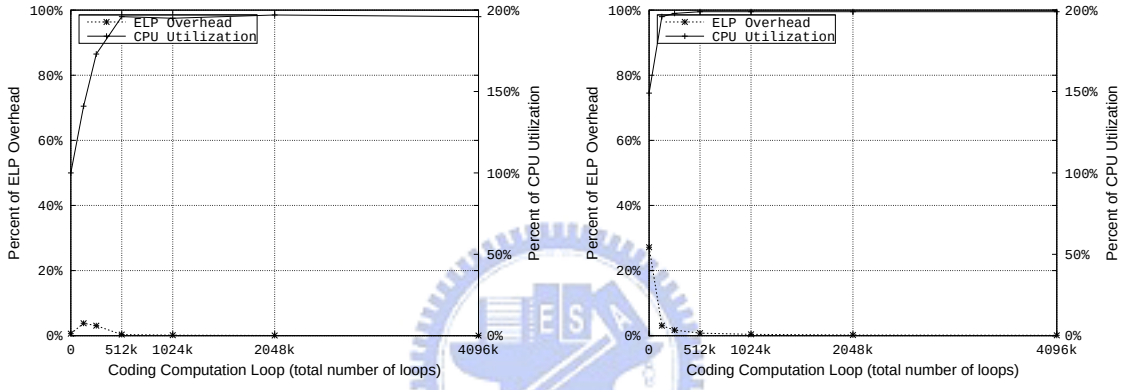
From Fig. 5.11(b), Fig. 5.13(b), and Fig. 5.14(b), one sees that the CPU utilization are close to 150% and the ELP overheads are about 30%. It means that the measter thread spent more time on finding the safe event set. Although the CPU utilizations are close to 150%; however, the ELP overheads are so high that the achieved performance speedups are not too high. This is because the CPU utilizations increased is canceled out by the effect of the increased ELP overhead.

From Fig. 5.12(a) and Fig. 5.12(b), one sees that the CPUs utilizations are close to 200% and the ELP overhead is very low. In these cases, good performances speedups will result.



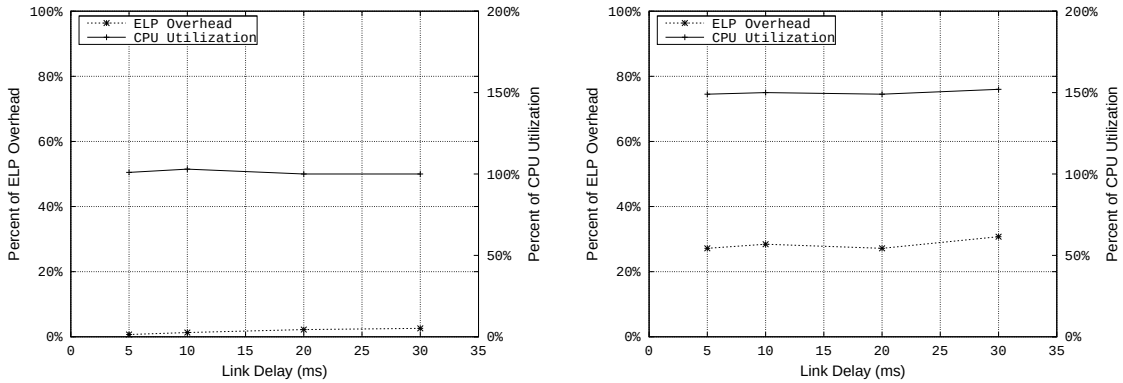
(a) UDP connections (b) TCP connections

Figure 5.11: ELP Overhead on a wired network: The network topology size is varied.



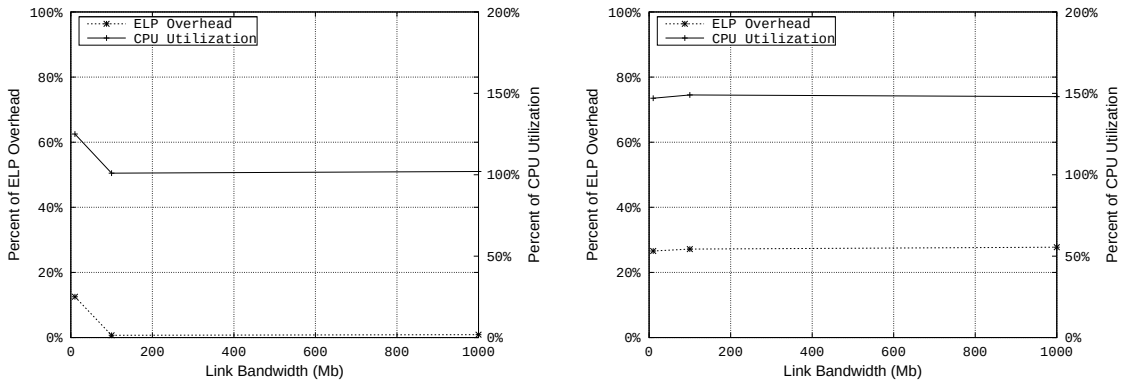
(a) UDP connections (b) TCP connections

Figure 5.12: ELP Overhead on a wired network: The coding computation loop is varied.



(a) UDP connections (b) TCP connections

Figure 5.13: ELP Overhead on a wired network: The link delay is varied.



(a) UDP connections

(b) TCP connections

Figure 5.14: ELP Overhead on a wired network: The link bandwidth is varied.

5.2 Traffic on Wireless Networks

5.2.1 System Paramemters

The common simulation parameters listed in Table 5.2 are used for wireless networks. In this table, the bold and italic strings are the default value when that system parameter is not varied. Most system parameters are the same as the wired network so that we briefly describe these system parameters as follows.

The network topology size is varied from 5, 6, 7, 8, 9, to 10. When this parameter is not varied, the default value for it is 10. The distance between a node and its neighboring nodes is 250m in the pattern shown in Fig. 5.15.

The bandwidths of all links are the same and are varied from 1Mb, 2Mb, 5.5Mb, to 11Mb. When this parameter is not varied, the default value for it is 2Mb. The link delay is computed based on the distance between two nodes and thus the link delay

Table 5.2: Common simulation parameters for wireless networks

Parameter name	**Value**
Network topology size	5, 6, 7, 8, 9, to ***10***
Link bandwidth	1Mb, ***2Mb***, 5.5Mb to 11Mb
Coding computation loop	***0***, 512K, 1024K, 2048K, to 4096K

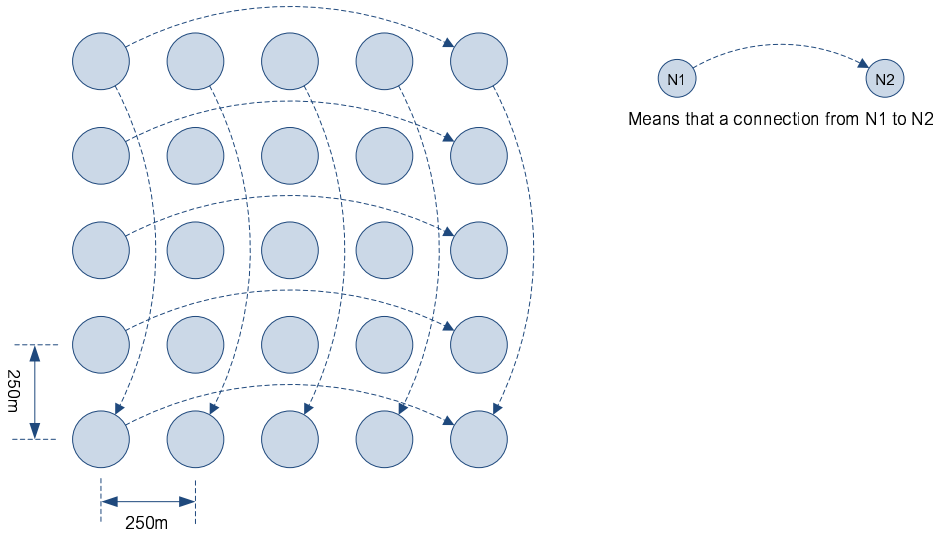


Figure 5.15: The connection pattern on a 5x5 grid wireless network

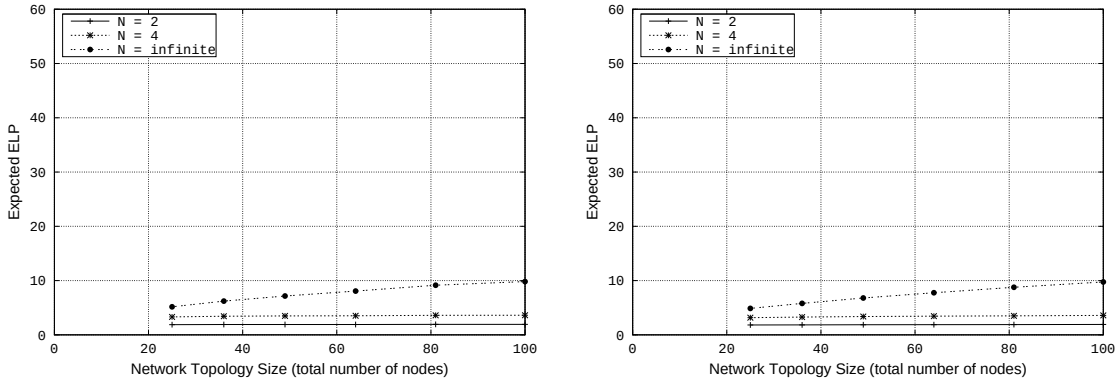
of a specific link between two nodes is the same for all simulated cases of wireless network. The packet size is the same as the system parameter used in the simulated cased for wired network. The coding computation loop is varied from 0, 512000, 1024000, 2048000, to 4096000. When this parameter is not varied, the default value for it is 0.

We also created UDP and TCP connections for each case and respectively measured the ELP degree and performance speedups and the connections pattern illustrats in Fig. 5.15. The traffic generator for UDP connections periodically sends packets at a constant bit rate and the interval time of it is 10us.

The simulated time for each case is 30 seconds despite whether it is for ELP degree evaluation or performance speedups measurement. On these dual-core systems, each simulation case usually took a time between 1 and 10 hours to finish, depending on the total number of packet events that need to be processed in a simulation.

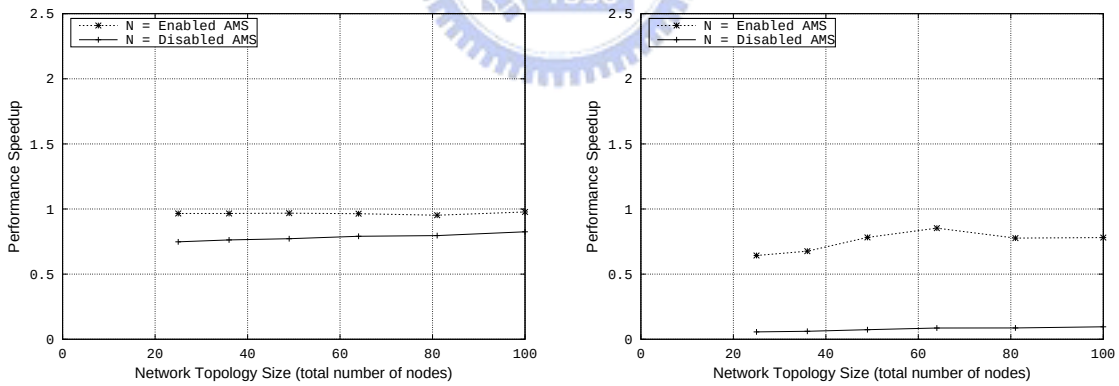
5.2.2 ELP Degree and Performance Speedups

We used the same version of simulator to compute the expected ELP of a simulation. Similarly, we compute three different versions of the expected ELP.



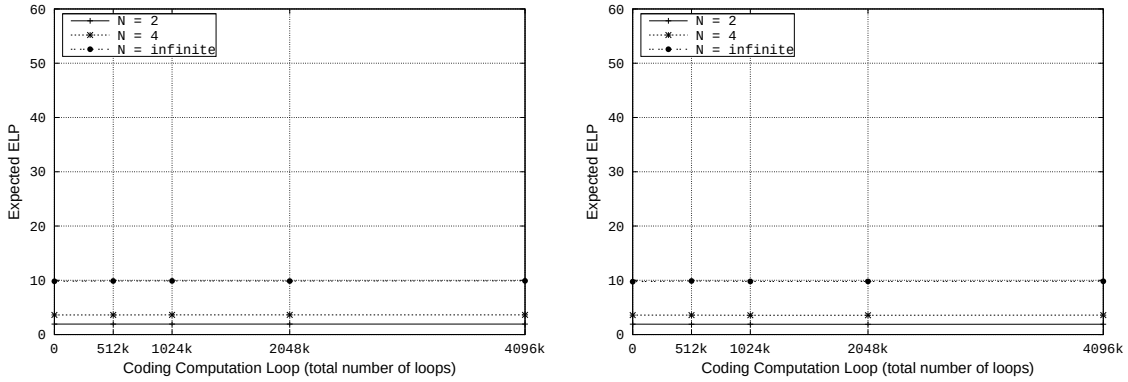
(a) UDP connections (b) TCP connections

Figure 5.16: Expected ELPs on a wireless network: The network topology size is varied.



(a) UDP connections (b) TCP connections

Figure 5.17: Performance speedup on a wireless network: The network topology size is varied.



(a) UDP connections

(b) TCP connections

Figure 5.18: Expected ELPs on a wireless network: The coding computation loop is varied.

From Fig. 5.16(a), Fig. 5.16(b), Fig. 5.17(a), and Fig. 5.17(b), one sees that similar results to the ELP degree for UDP or TCP connection on a wired network. Therefore, the performance speedups can be explained with previous treatments on a wired network under various network topology sizes.

From Fig. 5.18(a) and Fig. 5.20(a), one sees that the coding computations loop affects the expected ELPs and the achieved performance speedups minimally. We measured the ratio of local computation events to packet arrival events for the standard case as shown Fig. 5.19. We found that the ratio of UDP and TCP connections on wired networks is about 1.1 and 7.1, respectively. The ratio of UDP and TCP connections on wireless networks are about 700. Due to the above results, when the ratio is very large, the coding computation loop is affects the expected ELPs and the achieved performance speedups minimally. However, as Fig. 5.18(b) and Fig. 5.20(b) show, the coding computation loop still affects the achieved performance speedups clearly when the ASM mechanism is disabled, although it affects the expected ELPs minimally.

We create other simulated cases under different coding computation loop and the network topology size is fixed to 30. Other system parameters are not varied for this case and the performance speedups of UDP or TCP connections on a wireless network are also measured. The results of the performance speedups are shown in Fig. 5.21(a)

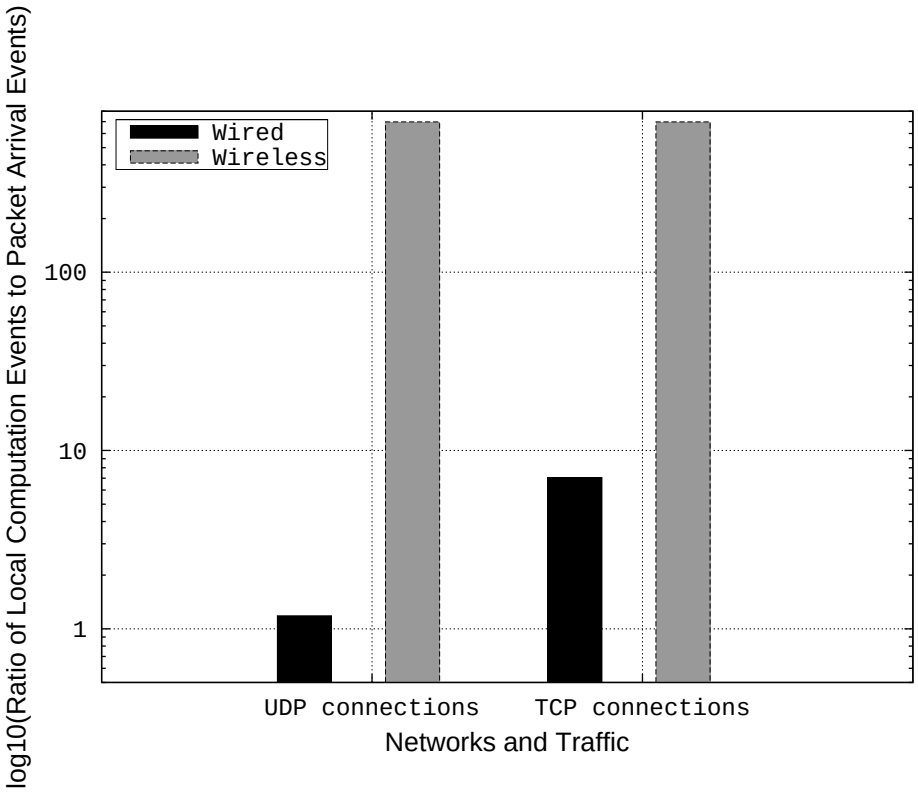
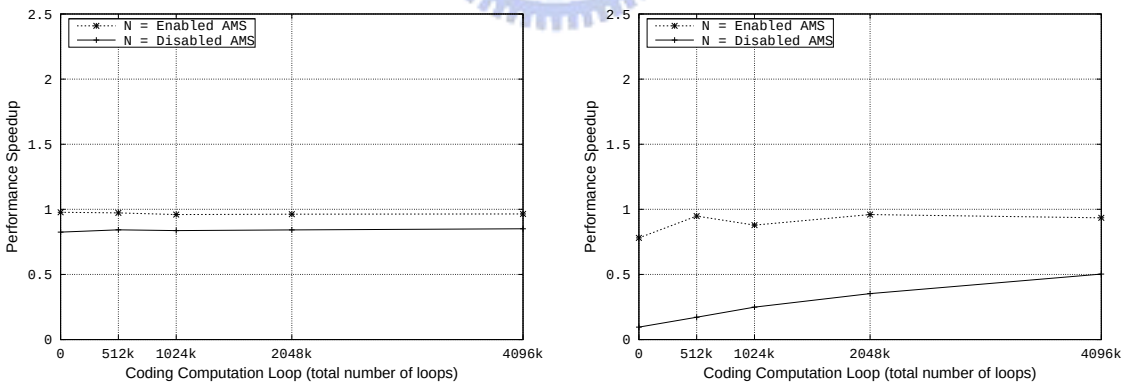


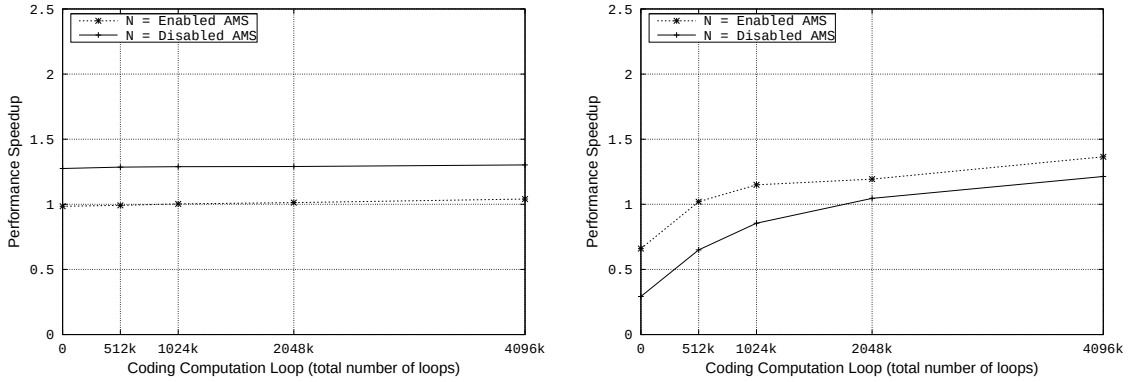
Figure 5.19: Ratio of local computation events to packet arrival events on wired and wireless networks



(a) UDP connections

(b) TCP connections

Figure 5.20: Performance speedup on a wireless network: The coding computation loop is varied.



(a) UDP connections

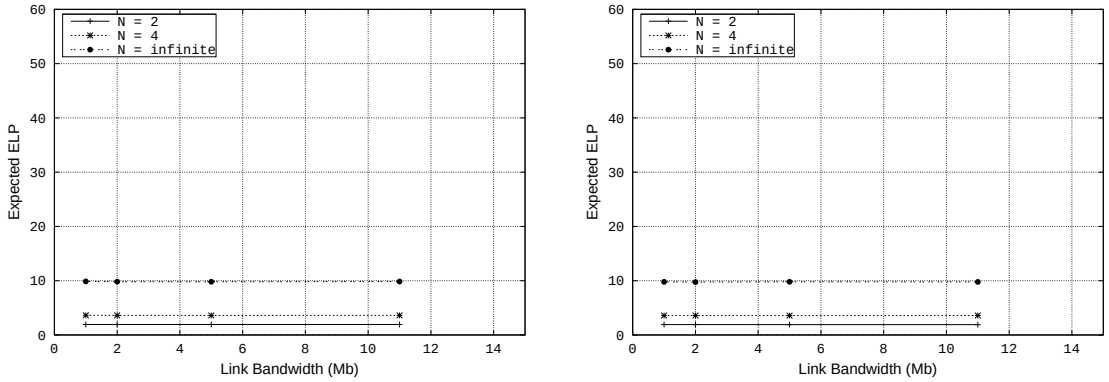
(b) TCP connections

Figure 5.21: Performance speedup on a wireless network: The coding computation loop is varied and the network topology size is 30.

and Fig. 5.21(b).

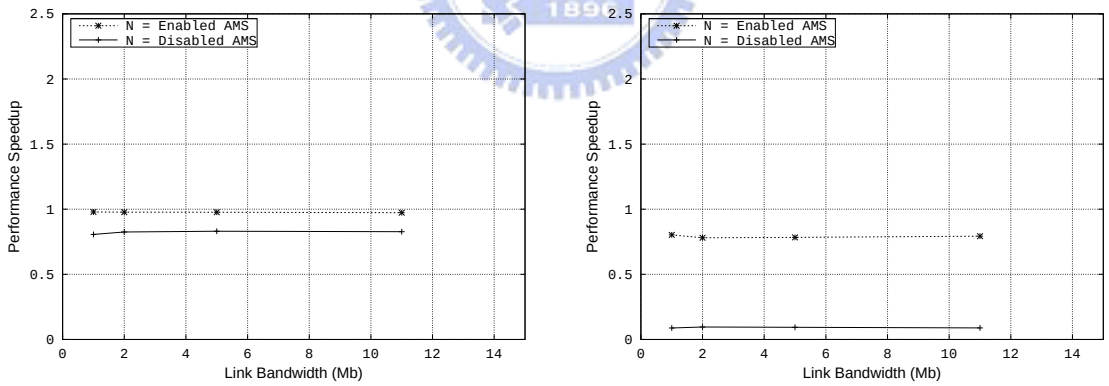
From these plots, one sees that the coding computation loop affects the performance speedups of TCP connections clearly and it affects one of UDP connections minimally. The average computation time of each event with TCP connections is smaller than the one with UDP connections. Therefore, the effect of the coding computation loop for TCP connections is clearer than the effect for UDP connections. The effect of the coding computation loop for TCP connections becomes smaller and smaller as the coding computation loop becomes larger and larger as shown in Fig. 5.21(b). On the other hand, although the effect of coding computation loops is very little for UDP connections, good performance speedups of UCP connections can result.

From Fig. 5.22(a), and Fig. 5.22(b), one sees that the link bandwidth affects the expected ELPs minimally. One may think that, in this simulated case, the propagation delay is far smallest than the packet transmission time. Therefore, several simultaneous packets on the flight would not occur so that the event density on the time axis would not change. Similarly, the link bandwidth also affects the performance speedups minimally.



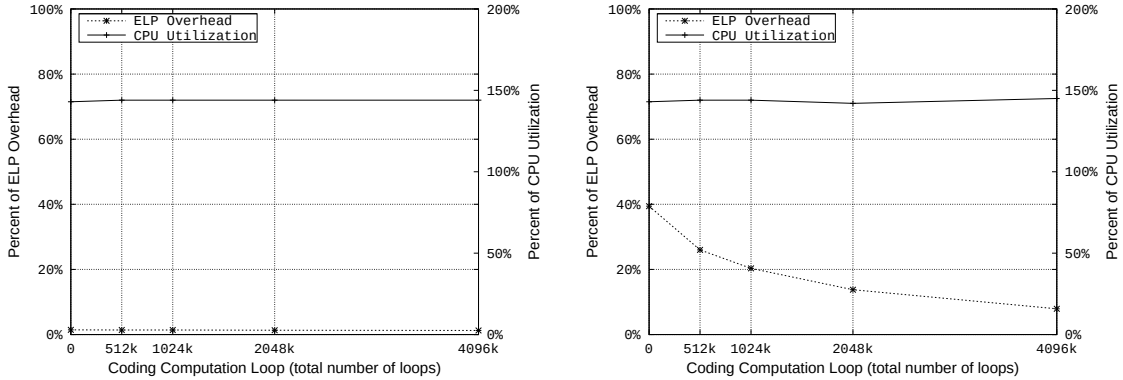
(a) UDP connections (b) TCP connections

Figure 5.22: Expected ELPs on a wireless network: The link bandwidth is varied.



(a) UDP connections (b) TCP connections

Figure 5.23: Performance speedup on a wireless network: The link bandwidth is varied.



(a) UDP connections

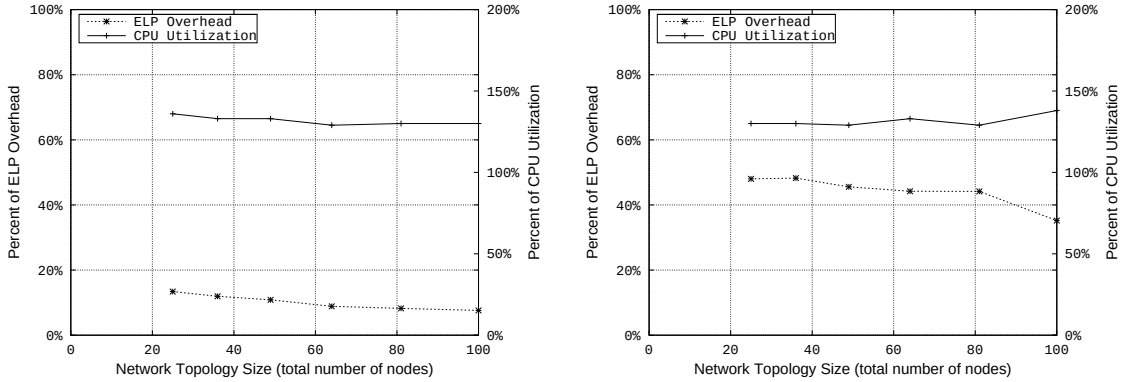
(b) TCP connections

Figure 5.24: ELP Overhead on a wireless network: The coding computation loop is varied and the network topology size is 30.

5.2.3 ELP Overhead in Percentage

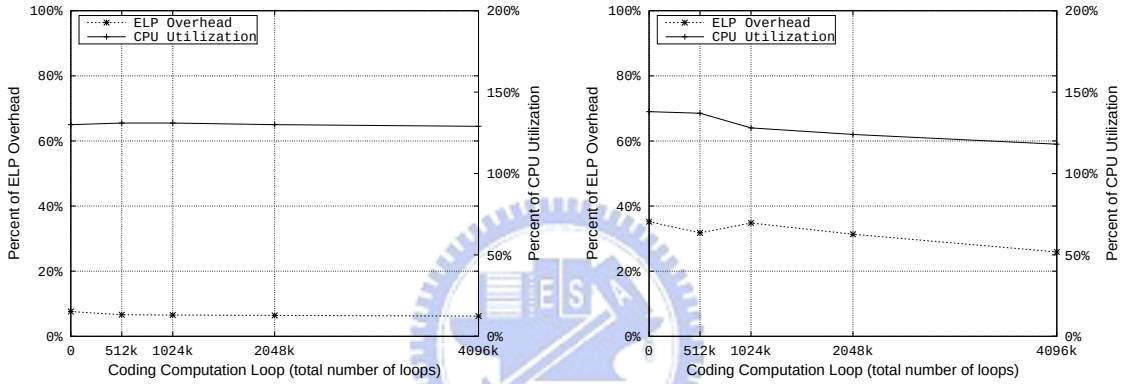
From Fig. 5.24(a), one sees that the CPUs utilizations are close to 145% and the ELP overhead is very low. In these cases, good performances speedups will result. As shown in Fig. 5.24(b), the smaller ELP overhead, the better performance speedups will result.

From the rest of the overhead plots, one sees that the CPU utilizations are close to 130% and the ELP overheads are over 10% and some ELP overheads are over 50%. It means that the worker thread spend more time on finding the safe event set. Although the CPU utilizations are close to 130%, however, the ELP overheads are too high so that the achieved performance speedups are not high. These results are similarly to Fig. 5.11(b).



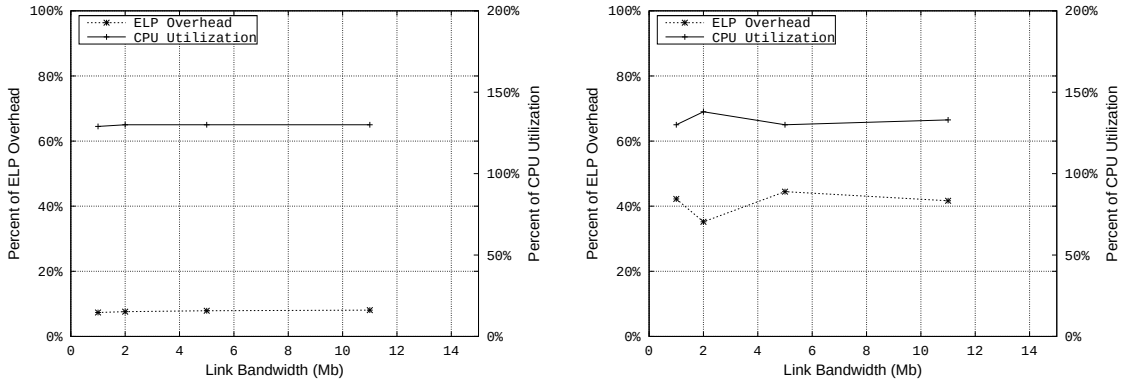
(a) UDP connections (b) TCP connections

Figure 5.25: ELP Overhead on a wireless network: The network topology size is varied.



(a) UDP connections (b) TCP connections

Figure 5.26: ELP Overhead on a wireless network: The coding computation loop is varied.



(a) UDP connections (b) TCP connections

Figure 5.27: ELP Overhead on a wireless network: The link bandwidth is varied.

Chapter 6

Future Work

- **Improving the AMS Mechanism**

  The AMS mechanism can automatically switch the simulation operating mode to the sequential mode when it detects that the computation time of the ELP overhead is greater than the average computation time of events. The current design is not smart enough to reduce wasted CPU cycles. Therefore, when a simulated case is very small, the performance speedup is not good enough.

- **Supporting the Mobile Network**

  In our implementation, the ELP-version of the NS2 network simulator is not capable of supporting mobile networks. There are more situations that should be considered for computing path lookahead in the mobile network. The location of each node may be different at any given time because it can move itself in the mobile network. In this situation, the master thread needs to re-compute the minimum path lookahead when it wants to find safe events. However it may increase the ELP overhead.

- **An Mechanism for Dynamically System Parameters Regulating**

  There are many system parameters, such as the MP value, the sleep timeout of worker threads, ..., etc. in our implementation. Currently, these system parameters are fixed. They may need a smart enough mechanism for dynamically

regulating them. Of course, this design may increase the ELP overhead. If this design is smart enough, the advantages may be greater than the disadvantages of this design.

- **Evaluating the Performance on N-core system**

  The simulator using the ELP approach is capable of executing on a N-core system and it is worthy to evaluate the performance speedup on a N-core system. However, the utility rate of the N-core system depends on the implementation of operating system. In addition, the system bus in hardware is a bottleneck that may affect the utility rate of a N-core system. We expect that the performance speedups cannot rise as linearly the number of cores is increased.

- **Applying the ELP Approach to NCTUns**

  The NCTUns network simulator [14] is also a network simulator and its architecture is very different with NS2. It is worthy to apply the ELP approach to NCTUns and evaluate the performance speedups of NCTUns using the ELP approach.

Chapter 7

Conclusion

We propose a novel and general parallel simulation approach and apply it to the NS2 network simulator on modern multicore systems. This approach exploits the event-level parallelism. The ELP approach requires only minor modification to the event processing loop of a network simulator to support the simple ELP thread architecture. As for the main body of a network simulator composing of various protocol modules, a simulation user need not learn an unfamiliar parallel simulation language and concepts to rewrite them for parallel execution.

In this thesis, we present the design and implementation of the ELP approach. We measure the degree of ELP under various network conditions and show that there are abundant ELP for parallel execution on modern multicore systems. Our experiment results show that when the average computation time of events is larger than the ELP overhead, the ELP approach provides good performance speedups under most conditions. On the other hand, when the average computation time of events is smaller than the ELP overhead, the performance speedup may be low. In this situation, according to our design, the simulator will automatically switch to the sequential mode to maintain its good performance by enabling the ASM mechanism.

Bibliography

[1] Loyd Case. *"Multicore Processors Transform at Rapid Pace"*. What's New @ IEEE in Computing, Vol. 8, No. 1, January 2007.

[2] Richard M. Fujimoto. *"Parallel and Distributed Simulation Systems"*. John Wiley & Sons, Inc, 2000.

[3] Kevin G. Jones and Samir R. Das. *"Execution of a Sequential Network Simulator"*. in the Proceedings of the 2000 Winter Simulation Conference, Wyndham Palace Resort & Spa, Orlando, FL, USA., 10-13 December 2000.

[4] Hao Wu, Richard M. Fujimoto, and George Riley. *"Experiences Parallelizing a Commercial Network Simulator"*. in the Proceedings of the 2001 Winter Simulation Conference, Arlington, VA, USA., 9-12 December 2001.

[5] OPNET Inc. http://www.opnet.com/.

[6] George Riley, Mostafa H. Ammar, Richard M. Fujimoto, Alfred Park, Kalyan PeruMalla, and Donghua Xu. *"A Federated Approach to Distributed Network Simulation"*. ACM Transaction on Modeling and Computer Simulation, Vol. 14, No. 2, April 2004.

[7] George Riley, Richard M. Fujimoto, and Mostafa H. Ammar. *"A Generic Framework for Parallelization of Network Simulations"*. in the Proceedings of the 1999 Modeling, Analysis and Simulation of Computer and Telecommunication Systems Conference, College Park, Maryland, USA., 24-28 October 1999.

[8] Daniel P. Bovet and Marco Cesati. *"Understanding the Linux Kernel"*. O'Reilly, third edition, November 2005.

[9] The Linux Kernel. http://www.kernel.org/.

[10] The Portable Application Standards Committee. http://www.pasc.org/.

[11] IEEE Std 1003.1. http://www.unix.org/version3/ieee_std.html.

[12] The LinuxThreads Library. http://pauillac.inria.fr/ xleroy/linuxthreads/.

[13] Ulrich Drepper and Ingo Molnar. *"The Native POSIX Thread Library for Linux"*. Computer Networks, Vol. 42, Issue 2, February 2003.

[14] S.Y. Wang, C.L. Chou, C.H. Huang, C.C. Hwang, Z.M Yang, C.C. Chiou, and C.C. Lin. *"The Design and Implementation of the NCTUns 1.0 Network Simulator"*. Computer Networks, Vol. 42, Issue 2, June 2003.