在事件層平行方法下 NCTUns 網路模擬器的效能量測

The Performance of the NCTUns Network Simulaor using the Event-level Parallelism Approach

研 究 生：黃文國　　Student：Wun-Guo Huang

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 2008

Hsinchu, Taiwan, Republic of China

中華民國九十七年六月

在事件層平行方法下 NCTUns 網路模擬器的效能量測

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

研究生：黃文國　　指導教授：王協源

國立交通大學

網路工程研究所

摘要

為了改進模擬大型、複雜網路環境所花費的時間，研究學者提出許多關於分散式平行模擬的演算法，它們主要將網路拓樸分散到不同的電腦上同步執行，讓模擬需要耗費的資源平均分散，以達到效能的提昇。現今的硬體配備已經發展出多核心的系統架構，記憶體的製成技術也愈來愈成熟，使得消費者可以用很便宜的價錢購買一台具有多核心高記憶體的電腦，也因為多核心電腦的普及和作業系統支援 SMP 架構，從前使用多台機器平行模擬的方式，漸漸的可以由一台多核心電腦完成。

在本篇論文中，提出一套新的分散式平行模擬方法，並將其應用在 NCTUns 網路模擬器。在本文裡，我們將詳細說明為了使用這套方法， NCTUns 網路模擬器在哪些層面必需稍做修改，以及量測 NCTUns 網路模擬器在各種不同網路條件下效能提昇的比例，並在最後提出這個方法未來的發展性和需要改進的方向。

關鍵字： 分散式模擬、平行模擬、多執行緒、多核心處理器、網路模擬器、NCTUns、ELP。

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

Student: Wun-Guo Huang Advisors: Prof. Shie-Yuan Wang

Institute of Network Engineering

National Chiao-Tung University

ABSTRACT

In order to decrease the time of simulating large-scale network, the parallel and distributed simulation method was advanced by the researchers. It slices the original network topology, and each one is independently executed by the different computer. Nowadays, the multi-core system already was developed by the hardware vendor and the manufacturing memory technology became more and more better. By theses reasons, Multi-core computers become more and more popular and the symmetric multiprocessing architecture already was supported by operating-system, so the original distributed and parallel simulation can be completed by a computer.

In this thesis, it presents a novel distributed and parallel simulation method, and applies it into NCTUns simulator. In the contents, we will detail how to design and implement in NCTUns simulator to support ELP and measure performance speedups of the NCTUns simulator which using ELP in various network conditions. Finally, the future developments and several improvement directions are proposed.

Keywords: distributed simulation, parallel simulation, multi-thread, multi-core, network simulator, NCTUns, ELP.

誌謝辭

首先要感謝的是實驗室裡的學弟、學長以及老師，在這兩年內，不斷的磨練，我從中學到了不少寶貴的經驗和研究的手法，同時也在完成份內工作後，提升了自我的專業知識。在課程學習上，由於同儕之間相互學習，使得我能更快的進入狀況，也可以更容易的融滙課程中所教授的知識。實驗室的學弟也在我最忙碌時幫我處理不少有關實驗室的事務，和有關實驗室開發軟體的問題，讓我在最後論文的撰寫上能無後顧之憂。

感謝家人的支持，令我能一路從小學一直讀到研究所碩士畢業，這是一條漫長的道路，途中曾碰到許多的挫折，不過有了家人的協助和鼓勵，這些阻力也轉化為我成長的動力，使我在求學的途中能順順利利的渡過各個關卡。

Contents

List of Figures

List of Tables

Chapter 1 Introduction

隨著網路環境的日益複雜，模擬一個龐大網路需要耗費的時間也愈來愈長。如何有效縮短模擬所需之時間，儼然成為一個新興的課題。在過去的時代裡，可以藉由使用較快頻率的 CPU 達到模擬時間的縮減。但是，過去的幾年內，對於提高 CPU 頻率的製程技術已經變的愈來愈複雜，從前使用的方法顯然已不再有效。

CPU 的開發廠商也慢慢正視到問題的存在，紛紛轉往多處理器架構在的研發，希望藉由多顆核心分散式處理的方法，達到效能提昇。實驗證明，這樣架構的發展下，效能上的確也有所成長。現今，已經有愈來愈多搭配多核心處理器的桌上型或是膝上電腦在市場上銷售。相信不久的將來，技術日加成熟的情況下，多核心處理器將永遠取代單核心處理器。

由於多核心處理器的普及，有效運用多顆核心的計算能力是相當重要的課題。然而，[1]指出，這對於任何原本執行在單顆核心上的應用程式要同時使用多顆核心資源是相當困難的一個問題，最主要的原因在於這些程式在執行時都只有使用到一顆 CPU 資源，它不需要考慮到不同核心間競爭的情況。為了能在多核心的系統中，獲得實質效能上的提昇，應用程式本身在設計上必需採用「多執行緒」架構[2]，使得不同執行緒可以同時使用不同 CPU 資源做運算。不過，採用多執行緒架構概念開發應用程式並不是一件簡單的工作，不僅無法保證核心個數以倍數成長時，效能也能擁有倍數提昇，且程式的維護修正上也比一般單核心困難的多。

除了硬體上的支援外，網路模擬器在發展的過程中，也有許多的學者提出各式各樣的方法，希望可以使用多顆核心資源增進模擬速度，縮減模擬時間。這些方法最終可以劃分成兩大類 – 保守法和樂觀法。無論使用哪一類方法，做用者都必需修改原有模擬器程式，從原有的架構中切出許多部份，使得每一個部份可

以同時執行，避免模擬結果錯誤的發生。保守法在實作上比較簡單，可是效能的好壞與 lookahead [3]有著莫大的關係。例如: 無線網路環境，鏈結的延遲非常短暫，相對可以同時執行的事件相對也不多。另一類的方法稱做樂觀法，它與保守法相比，有較好的效能提昇，不過在實作上也較保守法複雜的多。

不管使用上述的哪一種方法，都可能會存在效能較單核心系統低的風險。因此，我們提出一種新穎的方法，它不會改變原有循序執行的方式，但同時間亦保有平行運算的能力，我們將它稱之為「事件層平行模擬方法」 (Event-level Parallelism approach)。這個想法類似於硬體裡常用的「指令層平行方法」(instruction-level approach)。它利用不同指令同時平行運算的方式達到效能的提昇，這樣的作法不需考慮上層應用程式的特性，只需思考兩個指令間是否存在相依關係。

本篇論文章節的組成如下所述。在第二章中，我們將呈現有關傳統平行與分散式模擬方法的介紹、和 Linux 核心系統中重要資料結構的設計，包含: 行程描述器，核心排程器，核心鎖和核心執行緒。NCTUns 網路模擬器也在此章節中被介紹。在第三章中，我們將敘述事件層平行方法重要的元件，和討論尋找安全事件的規則。在第四章中，我們詳細說明在 NCTUns 架構下，事件層平行方法的設計及實作，並且提出一些在實做上的問題和解法。在第五章裡將呈現實驗得到的數據和效能分析。第六、七章，我們將分別提出事件層平行方法未來的擴充性和結論。

Chapter 2 Background

事件層平行方法 (Event-level Parallelism approach)與傳統的平行、分散式模擬方法 (Parallel and Distributed Simulation approaches)相似。它們使用多顆處理器系統減少為了模擬複雜且龐大網路環境所必需耗費的時間。在事件層平行方法下最困難的是如何有效提高事件平行度，及確保模擬結果的正確性，相同的問題也出現在傳統的模擬方法上。不同的是，事件層平行方法必需在支援多執行緒的作業系統和提供執行緒使用者函式庫的環境上運作。

在此章節中，將簡單介紹傳統的平行、分散式模擬方法，和使用這些方法的一些相關議題。接著介紹部份 Linux 核心資料結構，包含: 行程描述器結構 (task structure)、核心排程器 (scheduler)和多核心系統使用的同步機制 (spin locks); 除了核心結構的介紹外。章節最後會對這次論文中使用的模擬平台 NCTUns 做一個概括性的介紹。

2.1 Related Work

由於網路環境愈來愈龐大、複雜，模擬一個網路環境運作行為的時間也愈來愈長，為了解決模擬效率不彰的問題，將一個模擬的拓樸同時分散給不同的處理器執行的方法也紛紛提出。在[4]中，作者使用 Parallel discrete event simulation (PDES)的技術修改原有 NS2 網路模擬器 [5]的排程機制，使得它能平行運算事件，雖然效能上有提昇，但是這套方法並不適用在所有協定模組的網路環境中，它只能應用在 OSI-7 layer 建構出的網路環境底下。在[6]中，作者著重在無線網路環境底下效能的提昇並在文章中提出一項技術，可以容易的檢驗出事件是否可以平行運算，他從三個不同層面尋找能同時運算的事件，分別是 (i)event level, (ii) node level, and (iii) group level，不同層面由於 lookahead 大小的不同，可以找到平行運算的事件也不相同，不過這套方法仍需要將原有循序執行的網路模擬器切割成不

同的 logical process (LP)，不同的區塊透過 IPC 的方式進行訊息的交換，每一塊獨立執行的區塊在完成所有不會造成結果錯誤的事件後都必需等待其它部份的區塊也完成，才能繼續執行新的事件，這是因為不同區塊溝通的訊息可能還在傳送的途中，不存在任何的 LP 裡，造成模擬結果的錯誤，因此，使用 global barrier 的方式暫停已完成的 LP，使得效能的提昇有了限制; 再者，使用者必需具備平行運算的概念才能使不同處理器的負載都相等，達到最佳的效能，不過這也暗示了，網路愈大愈複雜，這項工作就愈加困難。

大部份 PDES 演算法不是破壞掉原有模擬器的架構，就是使用新的程式語言建構平行分散式環境，對於使用者來說，為了修改出一個可以平行執行的模擬環境必需學習相關的概念，但無形中這也成為使用平行模擬的一道門檻。在[7]中，作者開發出一套新的函式庫支援無線網路的同步模擬，不過這套函式庫並非使用大家所熟知的程式語言所撰寫，為了使用它來完成不同核心同步模擬執行，就必需花費時間去了解各種語法的使用。

2.1.1 Parallel and Distributed Simulation Overview

如何確保在多顆 CPU 上所運行的結果是正確的，是平行與分散式模擬最主要的議題。一般的循序模擬 (Sequential simulation)會從事件串列中挑選時間戳記最小的事件開始執行，所以，執行的過程中並不會出現擁有較大時間戳記的事件先被執行的情況發生。但在多核心架構下，這樣的案例卻有可能出現，因為同時間可能會有許多的事件一起執行，如果沒有判定哪些事件彼此之間會相互影響，就會造成模擬結果的不正確，這種狀況稱之為 「因果錯誤 (causality errors) 」，如圖 Figure 2.1 所示。

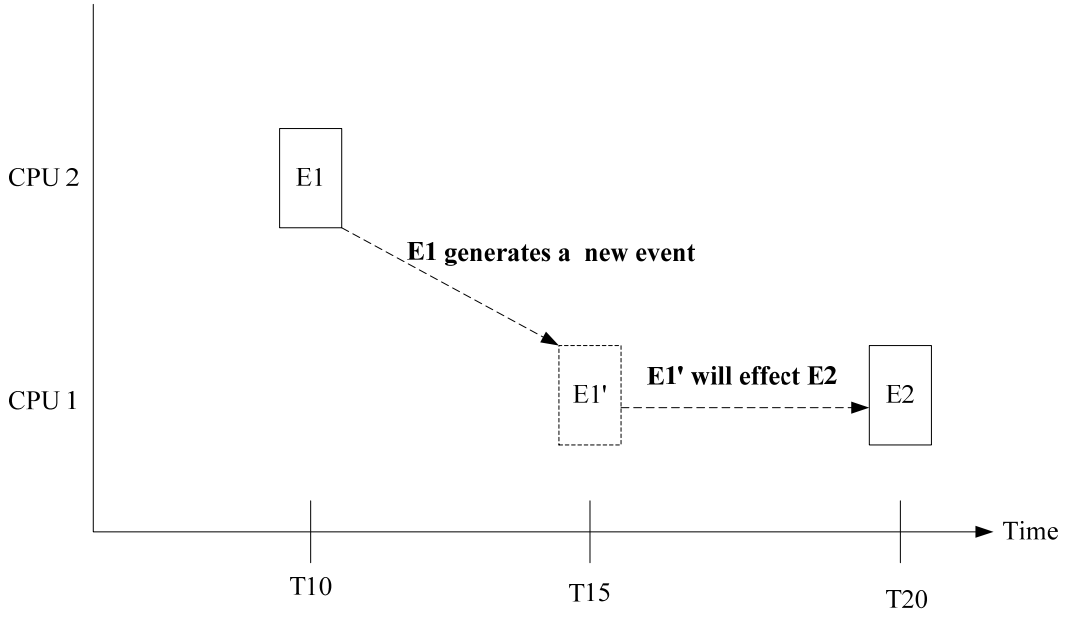


Figure 2.1 Causality Error Scenario

為了解決「因果錯誤」的問題，在平行與分散式的模擬中，會將網路拓樸切割出許多的「邏輯程序」 (Logical Process)，它們彼此為相互獨立的個體，每一個 LP 負責完成自己事件串列下的所有事件，為了確保每一個 LP 是在 time stamp non-decreasing ordering 的環境下完成各自負責的模擬事件，LP 間透過交換帶有時間的訊息來獲得「local causality constraint」(Events within each logical process must be processed in time stamp order)。雖然在 local causality constraint 下可以保證不會有「因果錯誤」的狀況發生，然而這個條件並非是必要的，違背「因果約束 (local causality constraint)」也不一定會造成「因為錯誤」發生。這是因為不同的兩個事件在相同的 LP 下彼此可能是相互獨立，即使兩個事件不按時間前後關係完成，也不會造成結果的不同。

對於使用平行與分散式方法模擬，同步機制非常重要。而為了達到同步，最有名的兩個方法為 – 保守法、樂觀法。使用保守法，每一個 LP 都遵循 local causality constraint，它們將被阻擋直到確定在事件串列中的元素都是安全的，才會開始執行。每一個 LP 所執行的事件都嚴格的照時間戳記的大小依序完成，所

以可以避免發生「因為錯誤」。另一個方法 – 樂觀法，則沒有完全照著時間順序執行事件，因此在使用樂觀法時，必需配合額外的 recover 機制，例如:「rollback」。

事件層平行模擬方法不同於前兩類方法，但對於「因果錯誤」問題的解法卻較前兩類方法容易，有關事件層平行模擬方法將在第三章有更詳細的介紹及解說。

2.2 Linux Kernel Data Structure for Task and Scheduler

NCTUns 是一套橫跨 user-level 和 kernel-level 的網路模擬器，且事件層平行模擬方法在 NCTUns 的應用上也與核心中的行程描述器及排程器有很大的關聯，因此必需先介紹行程描述器與排程器在核心中扮演什麼樣的角色，這對之後的章節會有莫大的幫助。

2.2.1 Task Data Structure in the Linux Kernel

行程代表一個正在執行程式的實體。當一個行程被創造出來，大部份都與創造它的父行程具有相同的屬性。邏輯上，它從原有的父行程中複制一份相同的 addressing space，並與父行程執行相同部份的程式。雖然它們是共享相同的程式頁面，不過在資料 (stack and heap)方面，它們彼此各自擁有一塊獨立空間。所以子行程對於程式中資料的修改，從父行程的角度來看並不會有任何的改變。

核心為了管理各個行程，使用行程描述器 (task structure)紀錄一個行程的所有資訊，包含行程優先權，行程使用哪一顆 CPU 上的資源或是被哪些事件給暫停，分配給此行程的記憶體位置等 …，詳細欄位如 Figure 2.2 所示。

行程根據其本身的狀態，會被作業系統將其置入到不同的佇列中。所有處於 TASK_RUNNING 狀態的行程都群聚在「執行佇列」 (run_queue)串列中。不同

狀態會有不同的處理方式。處於 TASK_STOPPED、EXIT_ZOMBIE 或 EXIT_DEAD 狀態的行程，不必連結在特定串列中。處於 TASK_INTERRUPTIBLE 或 TASK_UNINTERRUPTIABLE 狀態的行程，則被串在「等待佇列」(wait_queu)。

等待佇列有好幾種用途，特別是中斷事件的處理、行程同步以及計時。等待佇列代表的是一組休眠中的行程，當某個條件為真時，就會被核心喚醒。由於事件層平行模擬是在多顆核心中運行，為了達到各個核心的同步，修改 NCTUns 核心程式時，會在部份的程式碼中加入等待佇列的機制，這將在之後的章節會做更詳細的介紹。

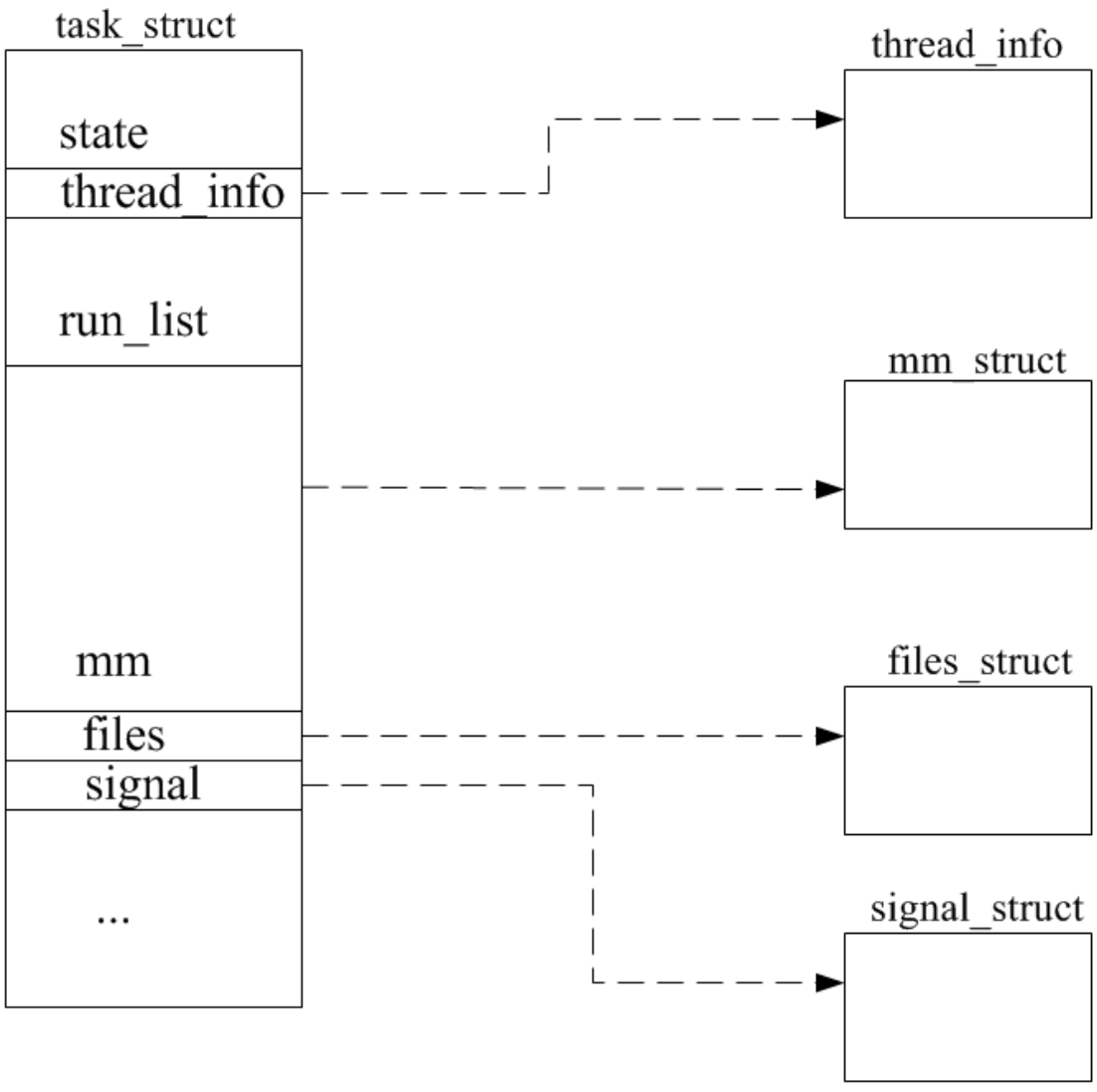


Figure 2.2 Task Data Structure

2.2.2 Scheduler Design in the Linux Kernel

核心排程器負責為每一個行程排定 CPU 時間，獲得 CPU 時間的行程才能使用系統內的資源。所以，排程器設計的重點在於如何公平且有效的分配 CPU 時間到各個行程。傳統的排程器在每次切換行程時都會掃描執行佇列內的所有行程，計算其優先權，並選出最佳行程來執行。這種方法最大的缺點在於花在找尋最佳行程時間的長短取決於可執行行程的數目。現在核心排程器為了解決上述的問題，會根據不同的優先權來選擇合適的行程分配給目前閒置的 CPU，具有相同優先權的行程會以雙向鏈結串連在一起，因此可在固定的時間內尋找到一支可以執行的行程。

如 2.2.1 節所敘述，行程會根據不同的狀態，被放到不同的串列中，排程器只對狀態處在 TASK_RUNNING 的行程感興趣，因此，其它不為 TASK_RUNNING 狀態的行程都無法獲得 CPU 時間。隨著優先權的不同，行程在核心中被劃分為兩類，優先權 0~100 屬於普通行程，而 101~139 是即時行程。

排程器根據不同行程優先權，分配給可執行行程的 CPU 時間長短也不同。普通行程目前使用的排程演算法稱之為完全公平排程(CFS)，它是使用紅黑樹的資料結構使得行程間能公平獲得相同的 CPU 時間，不會因為某些行程優先權太低而永遠拿不到 CPU 時間。即時行程使用的排程演算法不同於普通行程，當它搶到 CPU 時間，除非它自願將 CPU 讓出來，不然 CPU 會一直被某個即時行程給強佔。在 Linux 2.6.x 核心中，提出「行程先佔的概念」，它代表的是: 如果當前有一個行程的優先權高過目前執行中的行程，CPU 就會被優先權較高的行程給強佔。因此，新的核心架構設計下，即時行程的 CPU 使用權除了行程自願放棄外，另一個方式就是被其它的行程強佔。

2.2.3 Spin Locks

多核心的系統裡最為廣泛使用的同步機制就是上鎖 (locking)。不同的 CPU 可能同時在核心中存取一個共用資料結構或進入一個關鍵區 (critical section)，為了保護共同存取的資料，想要更改或是讀取共同資料的 CPU 就必須取得一個鎖。取得鎖的 CPU 可以存取關鍵區的資料，其它未取得鎖的 CPU 將在門外不斷等待，直到上一個取得鎖的 CPU 釋放鎖，等在門外的 CPU 才可以進入關鍵區內。

自旋鎖是 Linux 核心專門用在多處理器架構的同步機制。持有自旋鎖的行程會先檢查它手中的鎖是否有其它 CPU 正在使用，如果是，行程會自旋，也就是重覆執行一個指令迴圈，直到鎖被打開為止，相反的，如果並沒有任何 CPU 使用，它就會取得進入關鍵區的資格，並將自旋鎖的狀態設成上鎖。

一般而言，被自旋鎖保護的資料結構或關鍵區中，核心先佔功能是關閉的。就單處理器系統而言，自旋鎖並沒有什麼太大的用處，因為並不會有其它的 CPU 同時存取關鍵區的資料，但在使用事件層平行模擬方法的 NCTUns 網路模擬器中，自旋鎖機制就是一個必要的存在。

2.3 The Multi-threaded Evolution

如同第二章一開始的介紹，一個跑在多核心系統下的多執行緒程式需要作業系統和使用者執行緒函式庫的支援。我們將在接下來的小節中詳細介紹關於作業系統對於執行緒提供哪些支援，以及使用者執行緒函式庫的演進。

2.3.1 Thread Design in the Linux Kernel

執行緒是 CPU 排程的基本單元。一個行程並沒有限制可以擁有的執行緒數目，但是，一支行程通常代表的就是一條執行緒。因此，由同一支行程創建的執行緒將共享相同的記憶體空間。如 Figure 2.3 解釋，每一個方框代表一支行程。如 Figure 2.3 (a)所示，如果行程只擁有一條執行緒，這樣的程式稱為單一執行緒應用程式。

另一方面，如果行程內擁有超過一條執行緒時，如 Figure 2.3 (b)所示，則這樣的程式稱之為多執行緒應用程式。

在大部份傳統的作業系統，執行緒可以被分為兩大類，分別是使用者執行緒 (user thread)和核心執行緒 (kernel thread)。使用者執行緒只在 user-level 的程式中運作，它主要是透過使用者執行緒函式庫創建。另一方面，核心執行緒只存在 kernel-level。

在早期的 Linux 作業系統中，一支多執行緒的應用程式，無論是被創造、處理、排程都是透過使用者執行緒函式庫完成。多執行緒的應用程式裡的每條執行緒在核心中都只對應到一份行程描述器結構。因此，只要有一條執行緒正在存取 Linux 核心資源時，其它的執行緒將被暫停。

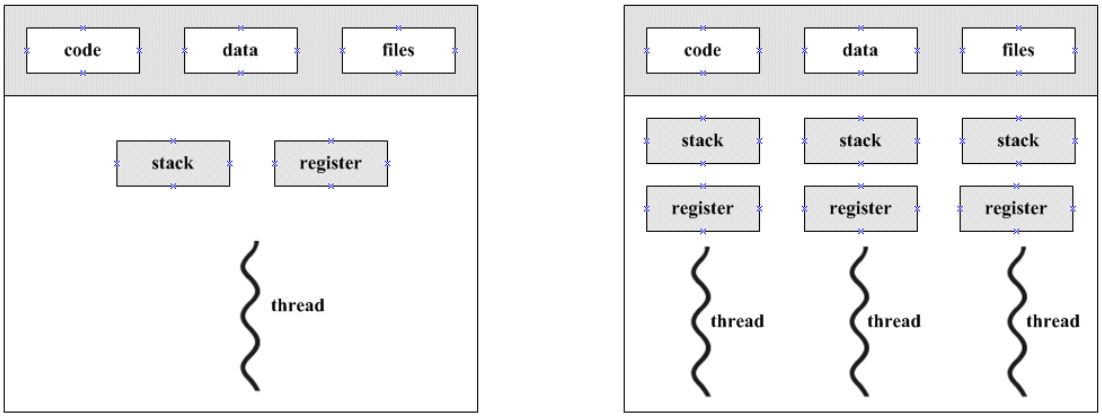


Figure 2.3 Threads Architecture

核心執行緒作用上與行程相似，因此，許多行程的特性，核心執行緒也具備。執心執行緒不同於使用者執行緒，它可以獨立的被核心排程。在 Linux 2.6 核心系統釋出之前，因為較舊的核心並沒有核心執行緒的設計，所以，過去行程與核心執行緒代表的是相同層面的結構，也就是說核心執行緒指的就是行程。從 Linux 2.6 核心開始，所有新的核心都已經有支援核心執行緒，它在 Linux 2.6 後的核心系統裡又被稱為輕量級行程 (LWP)。

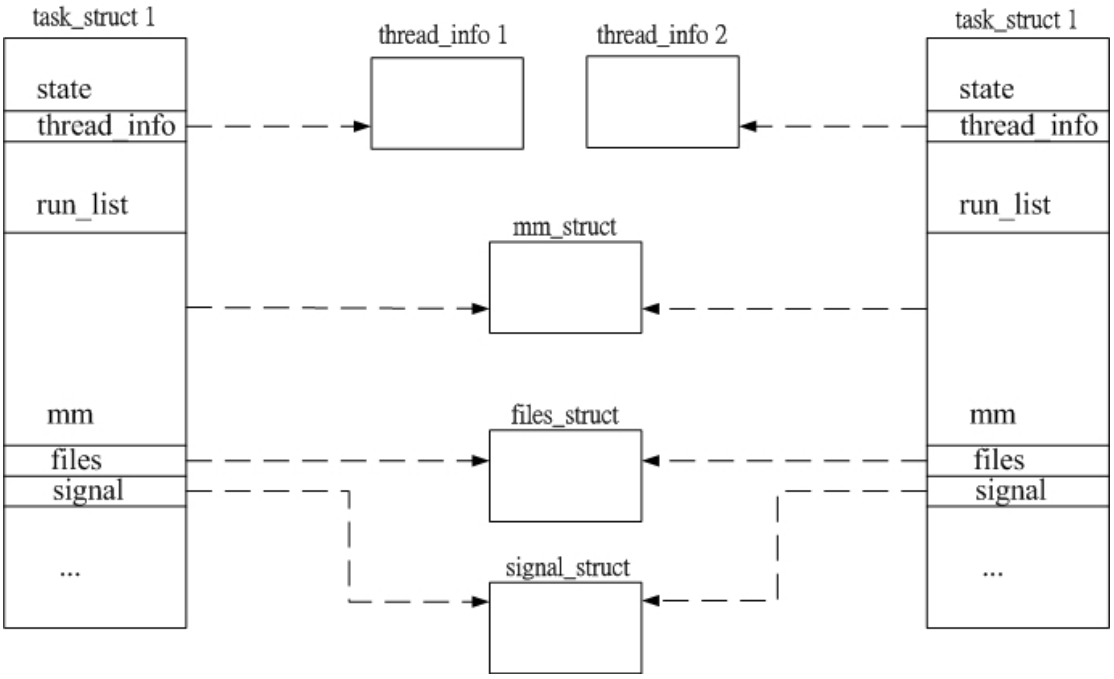


Figure 2.4 Light-weight Process Architecture

輕量級行程的架構，如 Figure 2.4 所示。每一個輕量級行程，都有一塊獨立的行程描述器空間，但是輕量級行程與它的父行程仍共用相同的核心資源。這樣的設計使的當某個輕量級行程對核心資源產生改變時，與其共用的核心執行緒將立即反應。

一個最簡單的多執行緒應用程式的設計，讓每一支使用者執行緒都關聯到一個輕量級行程。由於一個輕量級的行程可被核心獨立排程，因此，每一條使用者執行緒在核心中都可被視為獨立的個體。

目前的 Linux 作業系統已經支援 SMP 架構，並且整合新的架構到 Linux 2.6 的核心。如果有 N 顆 CPU，SMP 架構可以為每一顆 CPU 排定一個輕量級行程。

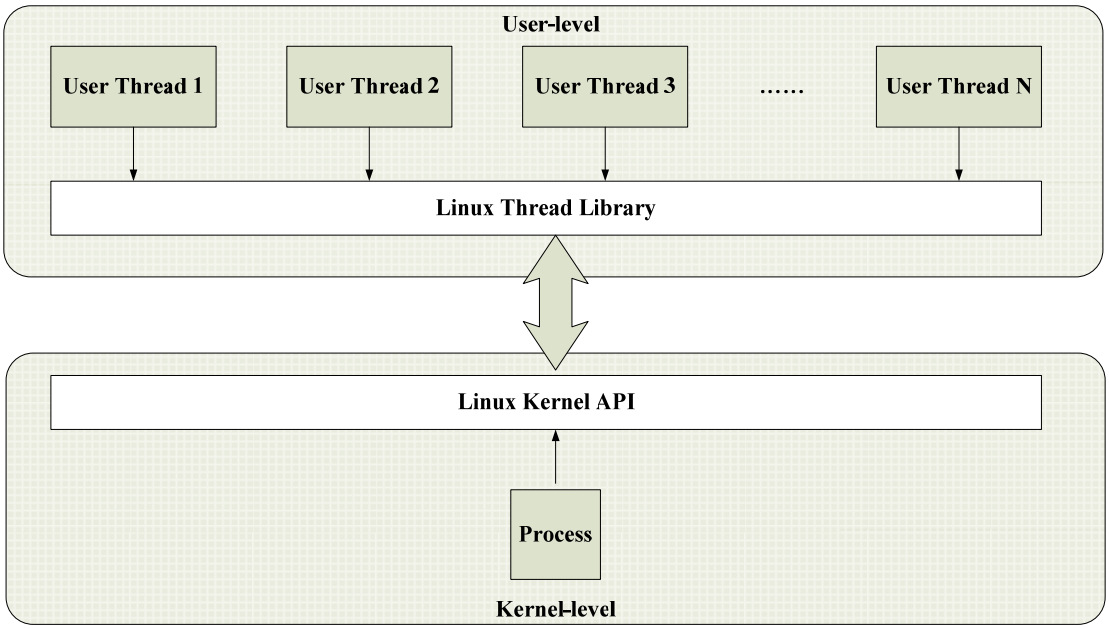


Figure 2.5 Nx1 Multi-Thread Model in LinuxThread Library

2.3.2 Thread Design in the User-level

在此小節中，我們將從最早開始的 LinuxThread，到目前比較廣泛使用的 NGPT 和 NPTL 使用者執行緒函式庫提出它們彼此的優缺點，及演進的過程。

The LinuxThread Library

LinuxThread Library 是最早被發展用來支援多執行緒應用程式的使用者執行緒函式庫。如 Figure 2.6 所示，它是以 Nx1 的模型做為多執行緒應用程式的架構。Nx1 模型代表的是所有的使用者執行緒在核心中都只對應一支行程。然而，這種作法並無法發揮多執緒應用程式的好處。因為，當某一條執行緒正在存取核心資源時，其它執行緒必需等待。

為了改進 LinuxThread Library 的缺點，兩個重新修改 LinuxThread Library 的計畫被提出，分別是 NGPT 和 NPTL。由於新的使用者執行緒函式庫開發已日漸成熟，所以對於 LinuxThread 的維護現在已經被中止。

The NGPT Library

發展 NGPT (Next Generation POSIX Thread)的團隊來自 IBM 的一群工程師。在[8]中有提到，NGPT 在效能上並不及 NPTL 突出，所以 NGPT 在 2003 年就已經被放棄繼續發展，取而代之的是 NPTL。目前有關多執行緒應用程式的開發，所使用的函式庫都已改為 NPTL。在 Linux 作業系統裡，NPTL 已經成為 POSIX-compliant library 的標準，而且它已經被整合進 GNU C library 裡。

The NPTL Library

NPTL (Native POSIX Thread Library)是 Red Hat 發展的一套使用者執行緒函式庫。它使用與 LinuxThread 函式庫相似的方法，藉由系統呼叫「clone」去完成多執行緒應用程式的實作。不同的是，NPTL 使用「clone」系統呼叫創建的是輕行程(lightweight process)，而非一般的行程。NPTL 並非完全是 user-level 的函式庫，它需要核心支援，使得每一支在多執行緒應用程式內的使用者執行緒都可以獨立排程。

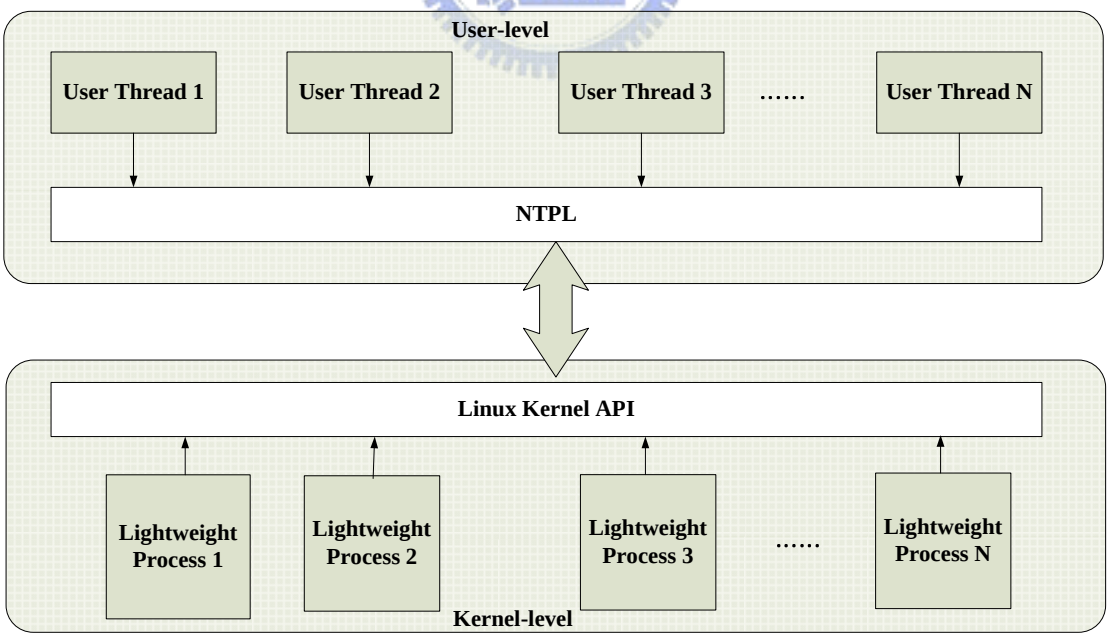


Figure 2.6 1x1 Multi-Thread Model in NPTL Library

NPTL 函式庫提供兩類不同的多執行緒應用程式的實作模型。第一類模型對每一條使用者執行緒在核心中都提供一支輕行程對應，如 Figure. 2.6 所示，這類模型被稱為 1x1 架構。使用 1x1 架構的多執行緒應用程式，在核心中都可以獨立被排程，所以，每條執行緒都可以同時存取核心資源。

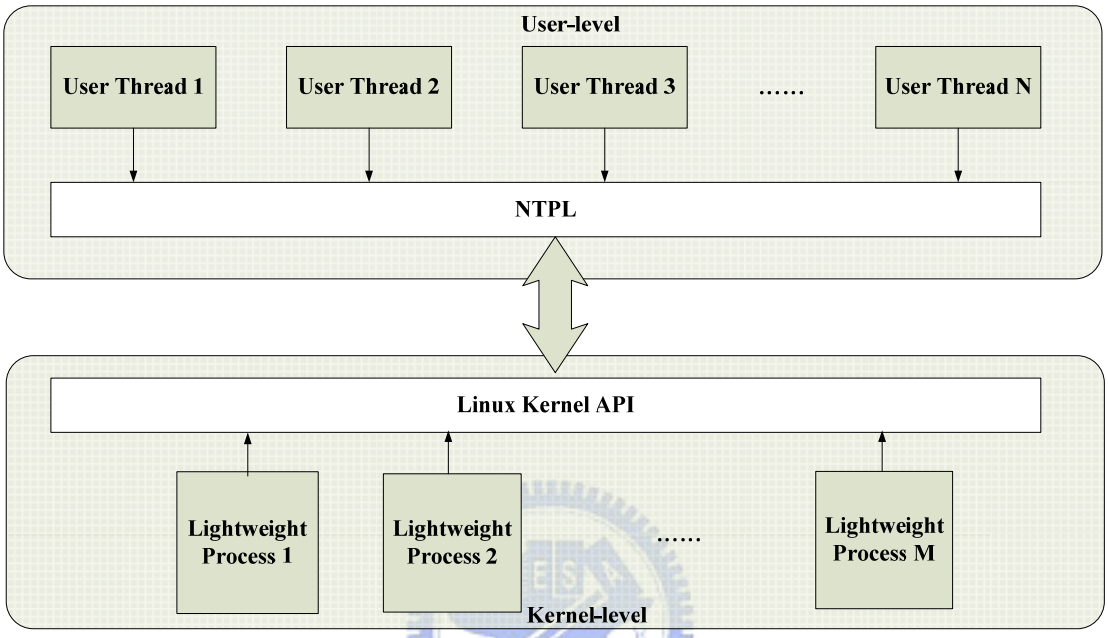


Figure 2.7 MxN Multi-Thread Model in NPTL Library

另外一類實作模型是 MxN 架構，如 Figure. 2.7 所示。它是將 M 條使用者執行緒對應到 N 支輕行程，所以，在此架構下的執行緒排程只在 user-level 實作。由於 MxN 的架構使得執行緒間的切換行為必需經由函式庫完成，造成使用 MxN 架構開發多執行緒應用程式的複雜度大大提高，因此，現今無論是哪一種 Linux 平台，關於多執行緒程式的實作都已經採用 1x1 架構。

2.4 The NCTUns Network Simulator

NCTUns 網路模擬器是採用模組化的設計 (protocol module based)，每一種型態網路使用各種協定模組建構，例如: 8023 有線網路由 interface、arp、fifo、mac8023、phy、link 模組所組成。因此，當某個新型態的網路出現後，只需實作其內部的協定模組，並更換進 NCTUns 網路模擬器裡，就可以開始執行新網路的模擬和

測試。它除了採用模組化的方式去實作網路底層協定的行為，更重要的是它採用一套名為 kernel-reentering simulation methodology 設計方法，如 Figure 2.8 描述，這套方法是由 NCTUns 的創建人王協源教授在 1999 年的美國哈佛大學時所設計的。由於這創新的方法，NCTUns 提供許多其獨特的優點，這些優點在傳統的網路模擬器中，如 ns2、OPNET，是沒辦法達到的。

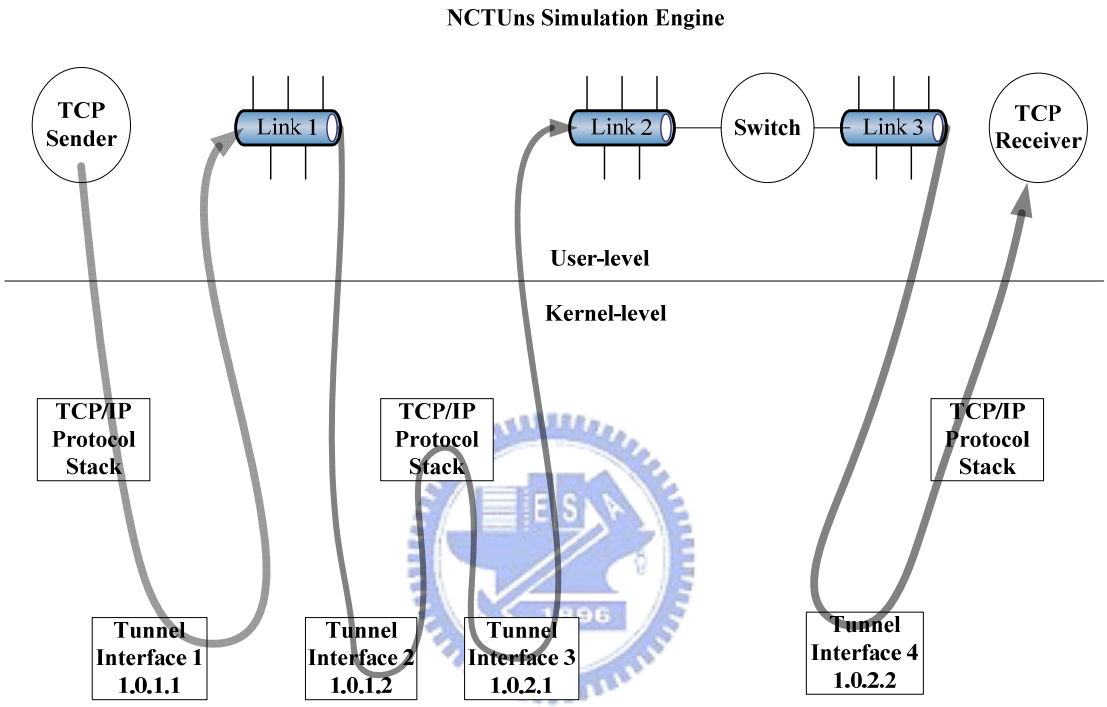


Figure 2.8 Kernel-reentering Simulation Methodology

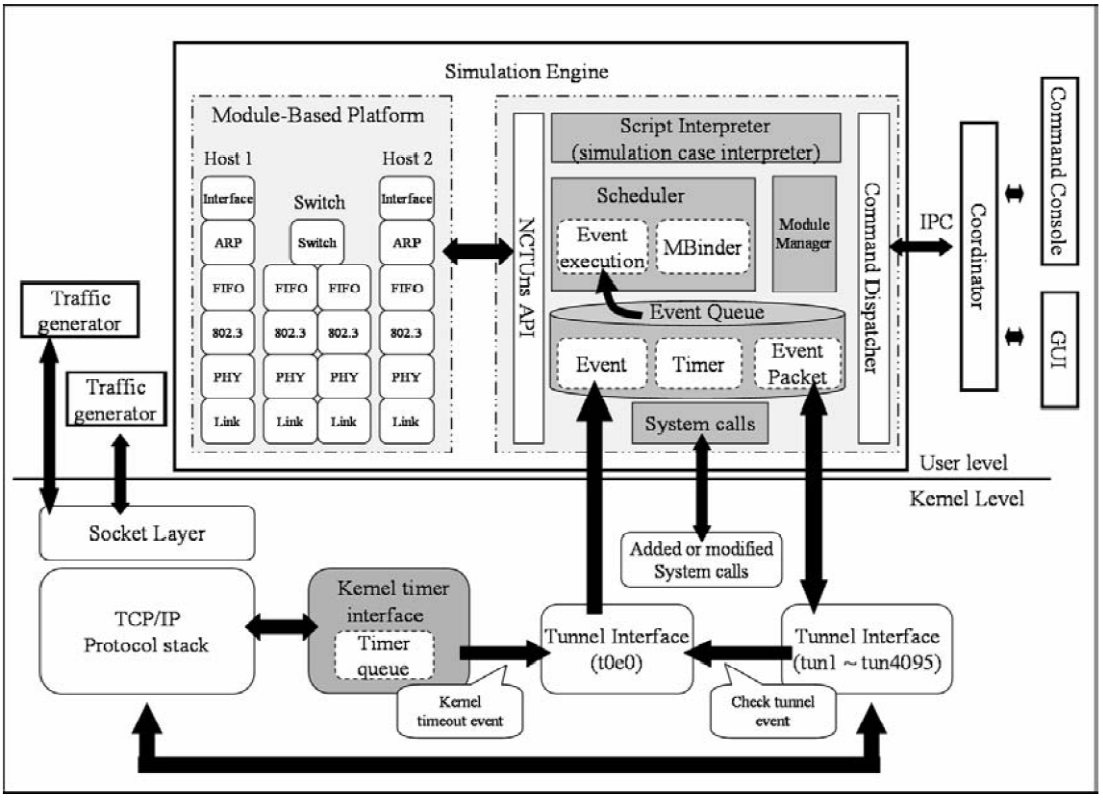
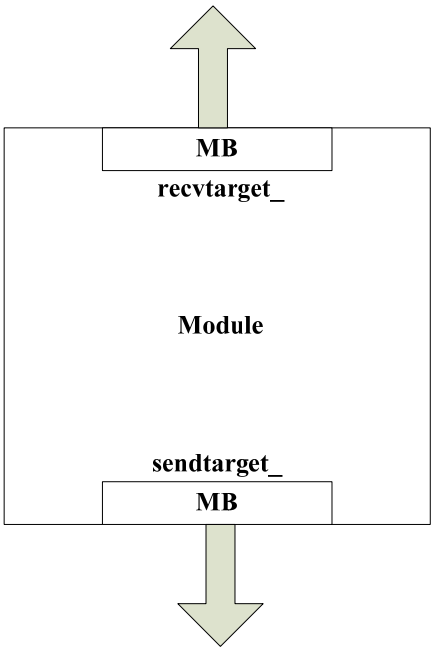


Figure 2.9 Simulation Engine and Module Relationship in NCTUns

NCTUns 網路模擬器分為三大部份，分別是模擬引擎 (Simulation Engine)、協定模組 (Protocol Module)，及 GUI Program。模擬引擎是一個 user-level 的程式，它是由許多複雜的功能所組成，包含模擬時間的管理，事件的排程，行程通訊，等…。它也提供許多 API 給協定模組使用，方便各個協定模組可以取得目前的系統資訊。模擬引擎和協定模組的組成關係，如 Figure 2.9 所示。一個網路節點由一串的協定模組所組成，NCTUns 的模組結構如 Figure 2.10 所示，研究開發者可以透過這樣的結構，實作出自己的協定模組，並且整合到 NCTUns。




```
int         init();
int         recv(ePacket_ *);
int         send(ePacket_ *);
int         get(ePacket_ *, MBinder *);
int         put(ePacket_ *, Mbinder *);
ePacket_    *put1(ePacket_ *, Mbinder *);
int         command(int argc, char *argv[]);
```

Figure 2.10 NCTUns Module Architecture

Chapter 3 Event-level Parallelism Approach

3.1 The ELP Architecture Overview

這一章節中，我們將介紹 ELP 整體架構。Figure 3.1 展示事件層平行模擬方法的結構。為了方便解說，接下來章節所展現的圖表都是從 four-core 的觀點出發。當然，它可以很容易的擴展至 N 核心的架構，只要將 worker thread 的個數從 3 變成 N-1 就能完成。

在事件層平行模擬架構中，主要由四個元素所組成。接下來我們將分別描述它們在事件層平行方法下扮演什麼樣的角色。

- master Thread

  主要的任務是找出目前哪些事件可在不同的執行緒上同步執行，並且不會造成「因果錯誤」發生。

- worker Thread

  執行由 master thread 找到的安全事件。事件執行過程中可能會產生新的事件被安插至全域事件串列 (global event list)。

- global event list

  執行過程中產生的新事件都先存放在此，等待 master thread 運算。全域事件串列有可能同時被 mater thread 跟 worker thread 存取。

- safe event list

  存放所有由 master thread 所決定的安全事件。放在安全事件串列所中的事件將在未來被 worker threads 所執行，且安全事件在不同的 worker thread 上同時執行並不會造成「因果錯誤」發生。同樣的情況，安全事

件串列也有可能同時被 master 與 worker thread 存取。

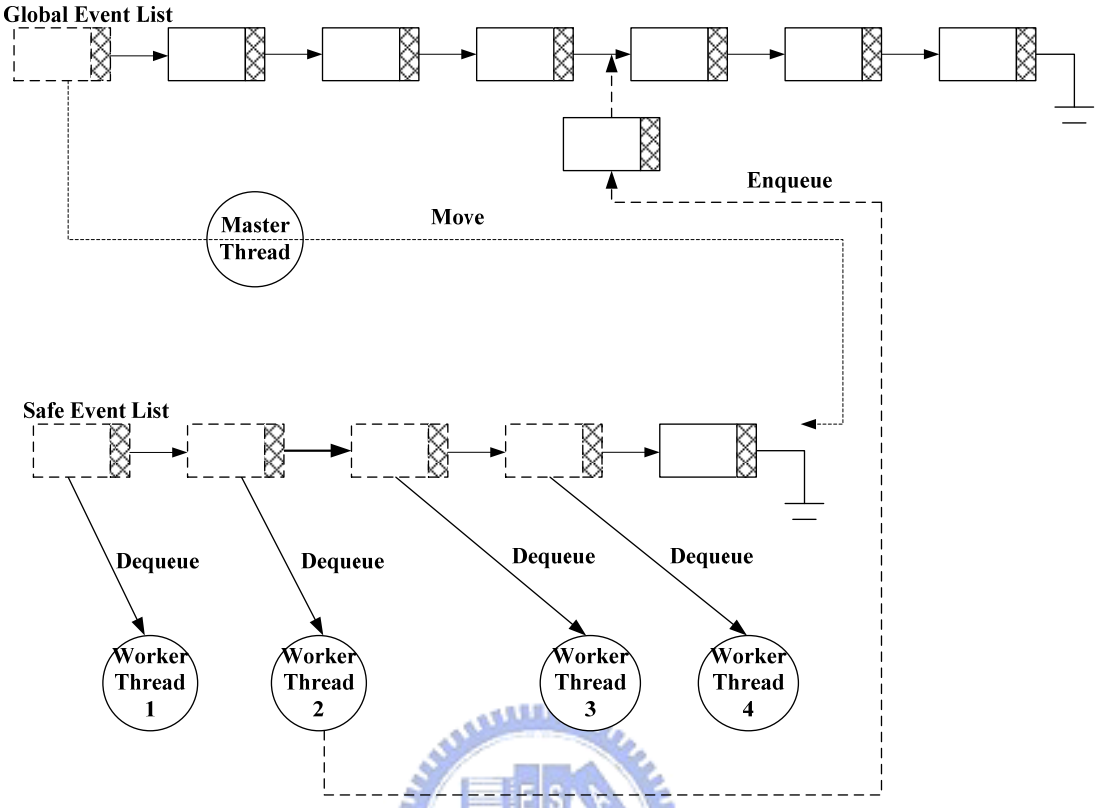


Figure 3.1 Event-level Parallelism Approach

對於四核心的系統來說，事件層平行方法會產生四條執行緒，其中一條執行緒為 master thread，而其它的執行緒則為 worker thread。master thread 主要負責安全事件的尋找，所找到的安全事件將被放進安全事件串列，由於這些被找到的安全事件彼此相互獨立，因此，不同 worker thread 平行執行，並不會造成模擬結果的錯誤。如 Figure 3.2 解釋，當 master thread 找到足夠多的安全事件後，它會叫醒因為沒有安全事件可做而正在沉睡的 worker thread。如果在目前的時間點上，找不到新的安全事件放到串列中，master thread 將沉睡直到當 worker thread 完成串列中的事件後，主動叫醒 master thread。當 master thread 醒來後，它會繼續找尋新的安全事件。Master thread 會重覆執行上述動作直到模擬結束。

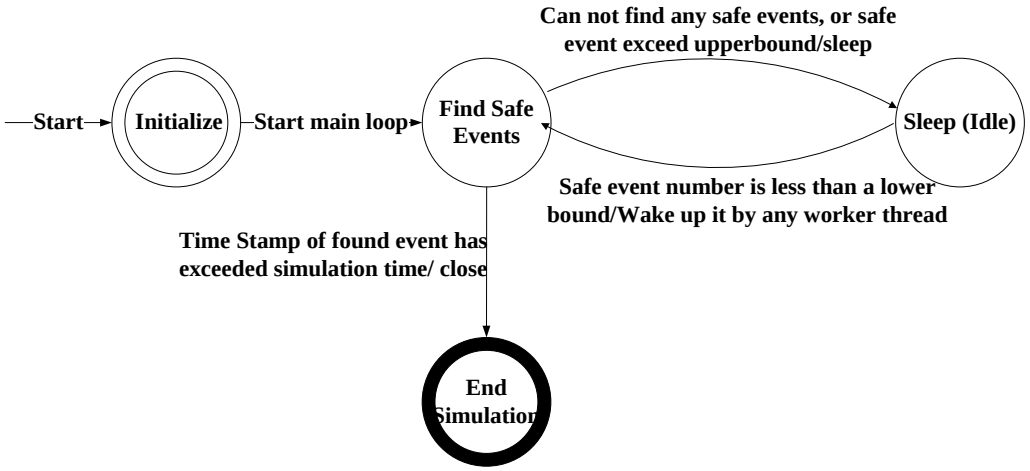


Figure 3.2 Master Thread State Diagram

在任何的時間點下，至少會有一條 worker thread 正在執行，因為事件層平行模擬是按照時間順序執行，擁有最小時間戳記的事件在整個模擬行進中，一定是安全事件。因此，假設目前只剩下一條 worker thread 正在執行安全事件，當它完成事件執行變成閒置狀態時，由於目前安全事件串列為空，所以 master thread 必需找尋新的安全事件放到安全事件串列中供 worker thread 執行，此時所找到的安全事件必為整個事件串列頭端，也就是時間戳記最小的事件。在新的安全事件加到安全事件串列後，worker thread 又可以開始執行，詳細的狀態圖如 Figure 3.3 描述。每當安全事件串列為空時，worker threads 會叫醒 master thread，所以 master thread 可以確保每次醒來時，安全事件串列裡並不存在任何的事件，它必需再從全域事件串列尋找安全事件。這也是為什麼事件層平行模擬較循序執行效能上可以提昇，或是，當所能找到的安全事件數不多時執行效能最多退回與循序模擬效能相同的因素。上述的想法可以很容易的達成，假設安全事件的執行僅透過單一的 worker thread 完成，其它的 worker thread 都轉為閒置狀態，而 master thread 只需把全域事件串列時間戳記最小事件放到安全事件串列，並不需要額外檢查其它事件與準備搬移事件的相依關係。按照上述的說法，事件層平行模擬方法將不會造成效能上低於循序模擬。

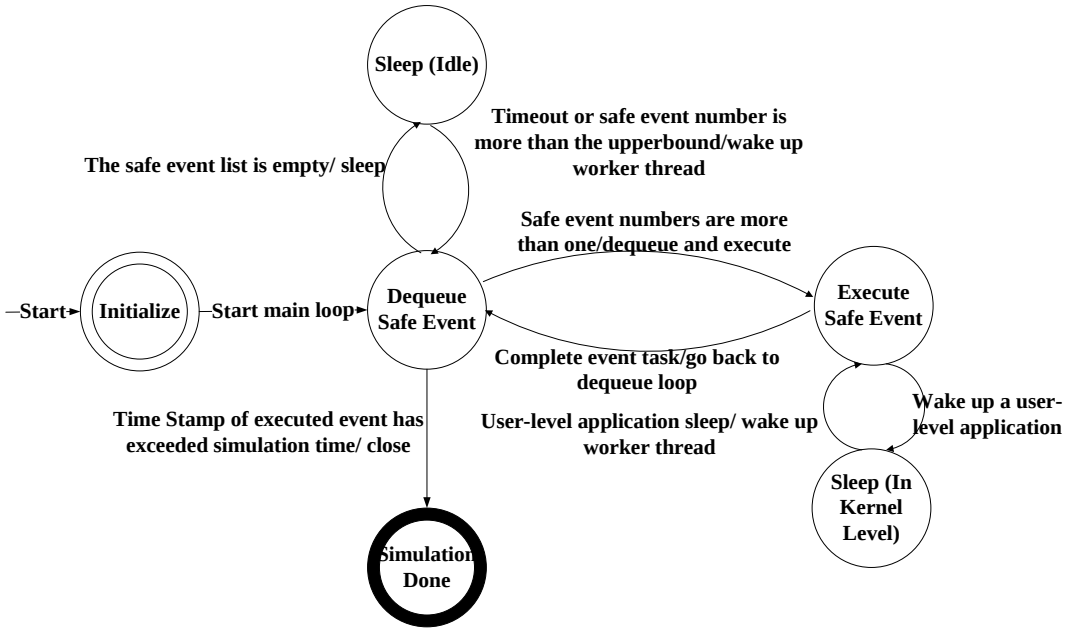


Figure 3.3 Worker Thread State Diagram

全域事件串列裡的事件都是未來將被執行的事件，這些事件是根據時間做為排序的準則，這些時間代表了事件被執行的時間，並且不會有亂序執行的情況發生。循序模擬器建構在單一處理器的系統上，在事件串列中的所有事件都是按照時間先後順序，有規則的執行，避免發生「因果錯誤」。在執行的過程中產生的新事件也將按照時間戳記被安插至全域事件串列。

事件層平行模擬方法，會隨著系統使用的處理器個數決定 worker thread 數目。假設目前有 N 顆處理器，會創出 N-1 條 worker thread 加上一條 master thread。只要 worker threads 能夠一直持續處理安全事件，沒有閒置的狀況發生，那麼隨著 CPU 個數的增加，效能上也可以線性提昇。master thread 重要的價值在於可否不斷找出足夠多的安全事件使得 worker thread 無時無刻都可從安全事件串列中找到事件進行運算。在 N-core 的系統下，只要 master thread 可以找到 n-1 個安全事件，讓每一條 worker thread 都有事件可以計算，效能上就能有 n-1 倍的上昇。

worker thread 每次完成安全事件的處理後，會檢查安全事件串列是否仍有足夠的安全事件可以讓 worker thread 執行，一般當安全事件數小於 CPU 個數減 1

時，worker thread 會叫醒 master thread 並通知它去尋找安全事件。Master thread 在找尋安全事件過程中，如果能找到比下限值 (worker thread numbers)高的安全事件數時，會叫醒沉睡中的 worker thread 重新開始執行，相反的，如果一直找不到夠多的事件數 (有可能目前 worker thread 正在執行的事件會影響安全事件的尋找)，master thread 會在嘗試一定的次數後，叫醒 worker thread，並等待 worker thread 完成目前的工作後，再一次叫醒 master thread。

因為 master thread 和 worker thread 都會同時存取事件串列與安全事件串列，為了確保資料結構一致性，必需在存取這兩個資料結構時使用 lock 的方式進行保護和協調。除了這兩個資料結構外，如果程式中仍有一些全域性的變數或資料結構有可能同時被不同的 threads 存取，則它們就需要使用 lock 保護。如果有些全域變數或資料結構並不會受到改變，而只是取出它們的值，則這些變數或結構並不需要 lock 來保護。由於每條 worker thread 可以同時執行安全事件串列內的安全事件，每筆安全事件所帶的時間都代表目前模擬所處的時間，所以正在執行事件的時間必需存放到不同的變數中，讓 worker thread 可以在執行的過程中，如果有需要用到目前模擬時間時，可以透過存取這個變數，獲得當前正確的時間。現階段每條 worker thread 都擁有一個屬於自己的資料結構，只需利用 thread ID 的對應就可以存取這個資料結構，所以並不需要 lock or unlock overhead。

3.2 The Event Relationship

在這小節中，我們將說明如何找尋安全事件提供給 worker thread 執行。能否找出更多的安全事件關鍵在於 lookahead 大小。因此，我們簡短說明安全事件與 lookahead 的關係。更詳細的說明可以參考[3]。假設一個網路模擬器正準備處理一筆時間為 T 的事件，而且與其它 logical process 透過訊息的交換得知下一個新的事件必需經過至少 L 單位時間才會被執行，那麼落在 T+L 單位時間內的所有事件都可被同時執行，並不會造成「因果錯誤」發生。這些可被平行運算的事件又稱之為「安全事件」。這個例子中所提到的 L 即是 lookahead。

在網路模擬環境底下，lookahead 可視為封包在鏈結 (link)上傳輸所花費的時間。這個值是訊號傳送時間 D 加上封包流出網卡所需傳輸時間 Tx 的總和。在這裡 D 是一個固定的值，而 Tx 則會與封包大小，線路頻寬有關。假設當前模擬時間為 T1 且目前正在執行事件 E1，E1 代表一個正準備從來源點傳輸至目的點的封包，如 Figure 3.4 (a) 描述，同時還有另一個帶有時間 T2 的事件 E2 即將在目的點執行，且 T2 大於 T1，只要可以確定 T1+D+Tx > T2，則可以保證 E1 將不會影響到 E2。這代表 E1 與 E2 都是安全事件，彼此可以平行運算，不會影響到模擬結果。相反的，如果T1 + D + Tx ≤ T2如 Figure 3.4 (b) 所述，那麼 E1 將會影響到 E2 事件的執行。因為在 E2 被執行前，有可能 E1 就改變了 E2 所處點的內部狀態，所以為了確保 E2 的執行一定是在 E1 完成後，必需保持 E1、E2 執行順序，故只有 E1 為安全事件。

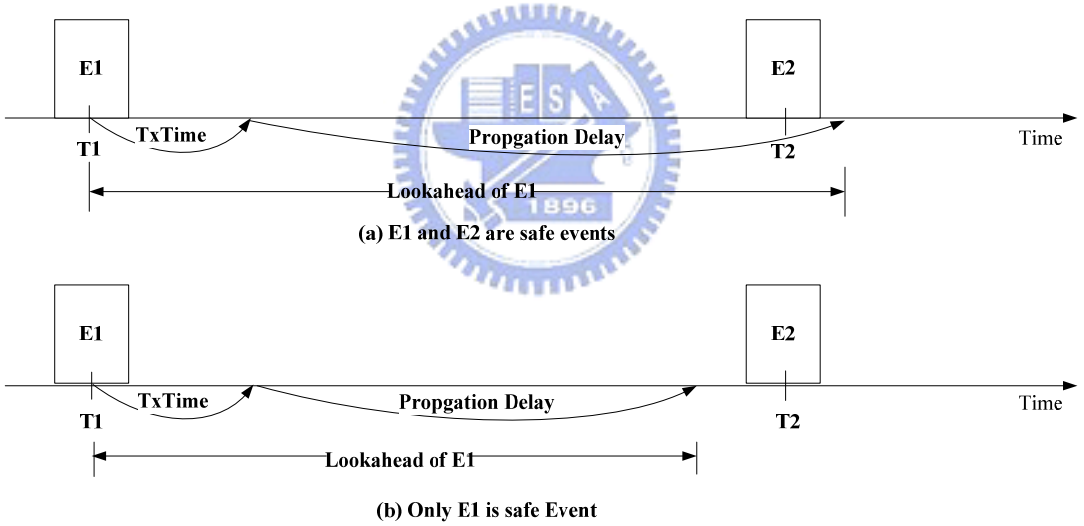


Figure 3.4 Relationship of Lookahead and Safe Events

當兩個節點沒有直接相連時，一筆較早被執行的事件 E1 可能會或可能不會影響到在另一個節點上較晚發生的事件 E2。接下來，我們將探討這樣的狀況，並提出一套規則來檢測是否 E1 會影響到 E2。在事件層平行模擬方法下，對於存放事件相關屬性的資料結構中，對每一筆事件都會儲存它是被哪個來源點 (SrcNID)所排程，作用在哪一個目的點 (DstNID)上。假設有一筆事件，它代表

某個封包從 Ni 被送到 Nj 的傳輸行為，那麼事件本身的 SrcNID 欄位將被設成 i 而 DstNID 欄位則被設為 j。封包被目的點接收後將改變其內部狀態 (e.g.，MAC 層從 MAC_IDLE 變成 MAC_RECV) 。另一方面，假設事件並非代表封包傳送，那麼這個事件一定是本地端計算的事件，它僅僅改變執行這個事件所在節點的內部狀態，並不會影響到其它節點的運作。對於一個在本地端運算的事件而言，假設這個事件在 Na 上執行，則它的 SrcNID 與 DstNID 會被設為相同的值，在這個例子中，它們將被設為 a。為了方便下面章節的探討，在此先對封包傳輸事件 (Packet Arrival Event)及本地計算事件 (Local Computation Event)做初步的定義，如 Figure 3.5 描述，圖中有三筆不同的事件，一筆為本地計算事件，另外兩筆為封包傳輸事件。本地計算事件 E3 在 N3 執行，所以 SrcNID 與 DstNID 被設成 3。另兩筆封包傳輸事件 E1 和 E2 分別是從 N1 送往 N2 與 N2 送往 N1。E1 內的 SrcNID 和 DstNID 被設成 1 和 2，而 E2 則被設為 2 和 1。在模擬過程中，這些值在固定後就不會再做更改。

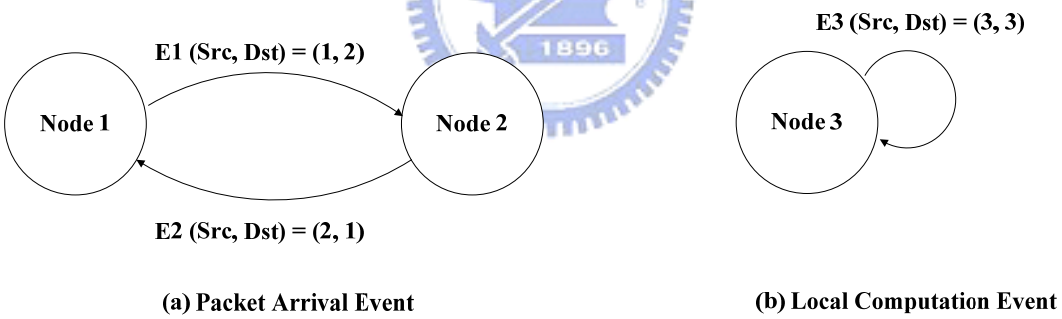


Figure 3.5 Event Type definition in NCTUns with ELP

在網路模擬器的架構下，不同網路型態都是由一連串的協定模組所組成，每一份協定模組代表的是各個不同網路運作行為。一筆代表封包從來源端送往目的端的事件，主要都是由協定模組中最下層的實體模組所排程，實體模組模擬一個封包在鏈結上的行為。試著去想像當一個封包從應用層送往下層模組傳輸的情境，此時上層模組會排定一個事件，代表應用程式產生出一筆新的封包。在這筆事件到達協定模組最下層的實體層模組之前，雖然它經過各式各樣的協定模組，但是

它仍在同一個節點上被執行，並不會影響到其它節點的狀態，所以它仍被視為本地運算事件。

接下來的小節，我們開始討論一筆事件的執行，是否會影響到另一筆在別的節點上執行的事件。

Rules	Condition
Rule 1	$SrcNID1 \neq SrcNID2$ and $DstNID1 \neq DstNID2$
Rule 2	$SrcNID1 = SrcNID2$ and $DstNID1 \neq DstNID2$
Rule 3	$SrcNID1 \neq SrcNID2$ and $DstNID1 = DstNID2$
Rule 4	$SrcNID1 = SrcNID2$ and $DstNID1 = DstNID2$

Table 3.1 Rules for Checking Safe Events

3.2.1 Packet Arrival Event

在這小節中，所有探討的案例都以傳輸事件做為基礎。假設目前有兩個傳輸事件 E1 與 E2，E1 由 SrcNID1 傳送至 DstNID1，E2 由 SrcNID2 傳送至 DstNID2。比較來源點 (SrcNID1、SrcNID2)與目的點 (DstNID1、DstNID2)的異同，可以得到四組不同的條件，如 Table 3.1 所示。首先思考的是規則 1 與 2，在這兩組條件下，只要 lookahead 的值夠大，可以很容易的去驗證 E1 並不會影響到 E2 的結果 (由前一節所討論的結果可得知)，如此一來就不用浪費時間在檢測事件相依性，大大提昇整體執行效能。

為什麼使用這四組條件來檢驗事件彼此相互獨立，我們將使用 Figure 3.6 解釋。在這張圖中，每一個節點 (node)都給定一個名字 Ni，i 代表的是節點的 ID，圖中的鏈結 (link)則表示為 Lij，i 與 j 代表鏈結上兩端點的 ID。每一個節點都與其相鄰的節點透過鏈結相連，且每一條鏈結都各自擁有一份 output buffer。圖中，Ea、Eb、Ec、Ee，和 Ef 代表的都是傳送封包的事件 (Packet Arrival Event)，而 Ed 則代表本地運算的事件 (Local computation Event)。每一筆封包傳送事件使用

箭頭指示符號表示它是由哪一個來源點傳送至哪一個目的點。例如 Ea 是由 Nf 傳給 Ne，Ec 則由 Nh 傳給 Ng。不同於封包傳送事件，本地運算事件則是使用一個自旋箭頭代表來源點與目的點相同。例如 Ed，它的來源與目的都為 Na。

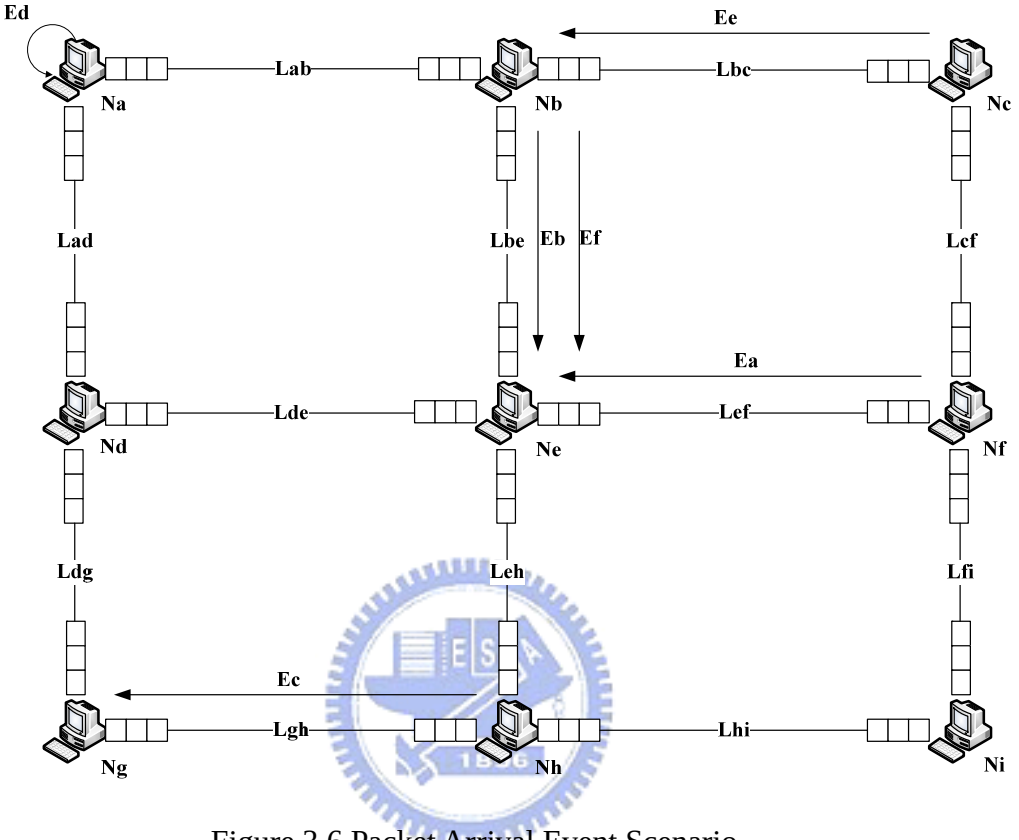


Figure 3.6 Packet Arrival Event Scenario

符合條件 3: SrcNID1 ≠ SrcNID2且DstNID1 = DstNID2的事件都被視為不安全的事件。在此用一個簡單的例子說明，試著思考 Ea 是否會影響到 Eb。假設 Ea 被執行的時間小於 Eb。因為一筆封包傳輸事件可能會改變接收點的內部狀態，所以很明顯的可以證明 Ea 是會影響到 Eb，為了確保模擬結果不會出現「因果錯誤」，必需確定 Ea 事件完成後，Eb 才可以開始執行。因此 Ea 與 Eb 不應該同時執行。

符合條件 2: SrcNID1 = SrcNID2且DstNID1 ≠ DstNID2的事件有可能成為安全事件，只需要 $DstNID_1$ 到 $DstNID_2$ 的時間夠長。在此用一個簡單的例子說明，試著思考 Eb 是否會影響到 Ed。假設 Eb 帶的時間小於 Ed，雖然 Eb 的目的端為

Ne 和 Ec 的目的端 Ng 並不相同，但是，如果存在一條 Ne 到 Ng 的路徑，Eb 有機會影響 Ed。當然，如果並不存在一條 Ne 到 Ng 的路徑，那麼 Eb 沒有任何機會影響 Ed。

為了能夠確定一個在 Nsrc 上較早執行的事件 E1 是否會影響到在 Ndst 上較晚發生的事件 E2，必需要計算所有可能從 Nsrc 到 Ndst 路徑需耗費的時間，從中找出一條需時最短的路徑。一段從 Nsrc 到 Ndst 路徑消耗的時間可以表示為 Nsrc->Ni->Nj->…->Ndst 各段鏈結花費時間的總和。最短路徑代表的是在 Nsrc 上較早完成的事件可以影響到在點 Ndst 上較晚發生的事件必需花費的最短時間總和。這是因為 Nsrc 也許會排定一個目的地為 Ni 的事件，當 Ni 收到這個事件，也許會在排定下一個事件，其目的地為 Nj，…，最後有可能 Nk 接放到這個事件，並排定一個事件到 Ndst。如果 Nk 排定到 Ndst 的事件所帶的時間小於 E2，且 E1 要比 E2 早執行，那麼 E1 就會影響到 E2。

由上述的說明可得知，要決定是否在 Ne 上的事件 Eb 可能會影響到在 Ng 上的事件 Ec，必需要去計算從 Ne 到 Ng 最短路徑所耗費的時間。它可以透過 all-pairs shortest path Dijkstra's algorithm 事先計算，並儲存所有可能最短路徑配對需花費的時間。假設事先計算得到 Ne 到 Ng 所需時間為 PLAeg，如 Figure 3.7 所示，Eb 的時間為 Tb 且 Ec 的時間為 Tc。如果 Tb+PLAeg > Tc，Eb 不可能影響到 Ec，且它們可以平行的運算。

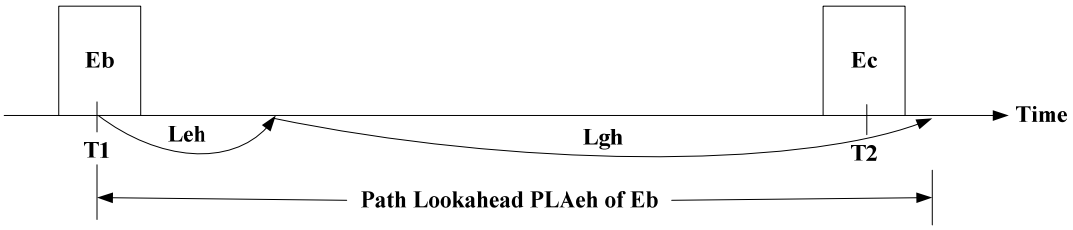


Figure 3.7 The path lookahead is the sum of all links lookahead on the path

符合條件 1: SrcNID1 ≠ SrcNID2且DstNID1 ≠ DstNID2的事件，絕大部份被認為是安全事件。驗證的方式與條件 3 相同。在此用一個簡單的例子說明這個條

件的可行性。試著觀察 Ea 是否可以影響到 Ec，假設 Ea 執行的時間小於 Ec。如同條件 3 所描述的，假設從 Ne 到 Ng 最短路徑所花費的時間總和為 PLAeg 而 Ea 的執行時間為 Ta，Ec 的執行時間為 Tc，如果 Ta+PLA27 > Tc，那麼 Ea 不會影響到 Ec，它們彼此可以平行的運算。

符合條件 4: SrcNID1 = SrcNID2且DstNID1 = DstNID2的事件，都是不安全的事件。用一個簡單的例子說明，試著觀察 Eb 是否可以影響到 Ef。假設 Eb 執行時間小於 Ef。雖然一張網卡同時間只能接收一個封包，但是仍可能在同時間內有許多的封包傳輸事件具有相同的 NIDsrc 與 NIDdst (ex: 它們被傳送透過相同的鏈結)，被排程至事件串列。這樣的例子，當鏈結有較大的延遲且高頻寬的條件下，就會出現。這會使得一個封包的傳輸時間小於鏈結造成的延遲，變成有許多的封包同時都在鏈結上傳輸。很明顯的，這些傳輸的封包到達接收端的順序必需被維持，所以它們彼此之間不應該同時的執行，否則將造成模擬結果的不正確。

3.2.2 Local Computation Event

這一節，我們將思考是否存在一筆在某個節點上較早執行的事件 E1，並不會影響到另一筆在不同節點上較晚執行的事件 E2。顯然的，E1、E2 彼此一定有一筆是本地計算的事件。對於一筆本地計算的事件，它的 NIDsrc 與 NIDdst 是相同的，所以考慮一筆本地運算的事件是否為安全事件，只需思考它的 NIDsrc 或 NIDdst 是否相同。

如同前一小節所提到的規則一樣，無論 E1 或是 E2 何者為本地計算事件，或者兩者都是，符合條件 DstNID1 = DstNID2 的事件都將被視為不安全的事件。我們將使用 Figure 3.8 探討DstNID1 ≠ DstNID2的案例。Figure 3.8 和 Figure 3.6 是相似的，在圖中 Ea 和 Eb 代表的是封包傳送事件，而 Ec 和 Ed 代表的是本地計算事件。

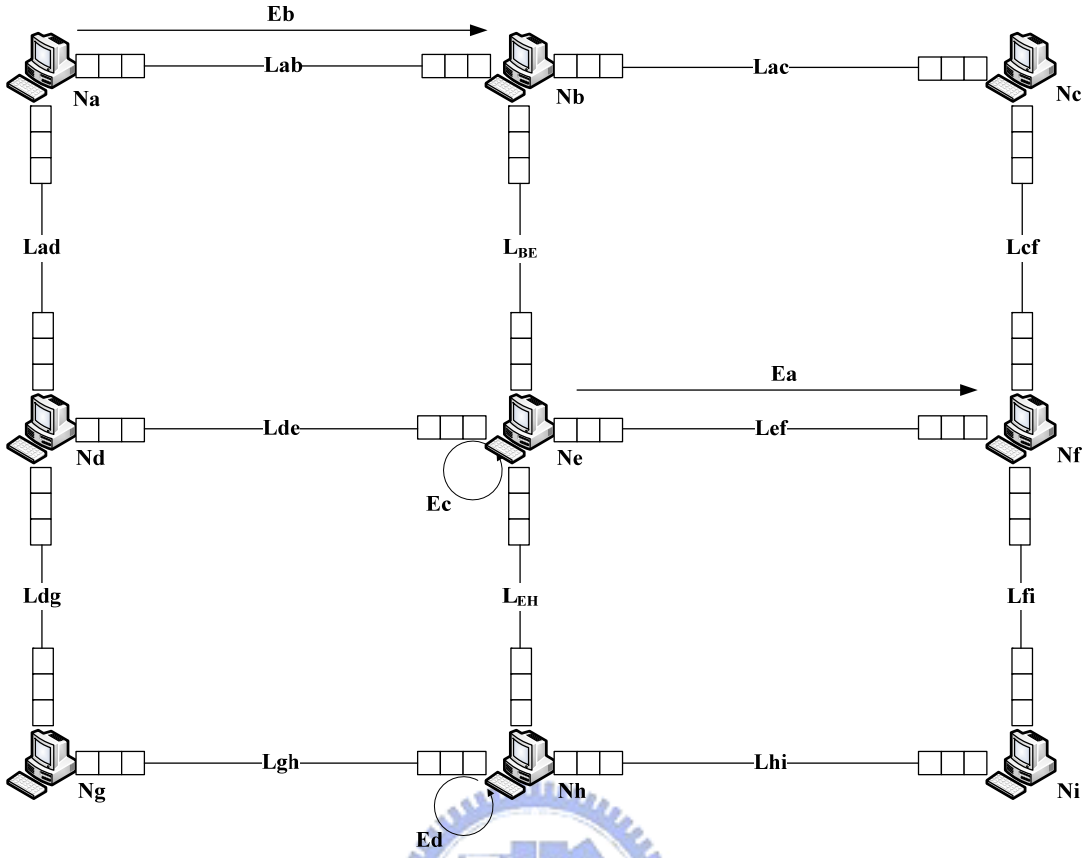


Figure 3.8 Local Computation Event Scenario

假設 E1 和 E2 是本地計算事件，很簡單地可以決定是否 E1 可能會影響到 E2。如果DstNID1 = DstNID2，E1 將影響 E2，因為它們執行在相同的點上。另一方面，如果DstNID1 ≠ DstNID2，E1 仍有機會影響 E2，透過一條 $DstNID_1$ 到 $DstNID_2$ 的路徑。一個簡單的例子將說明這樣的狀況，試著觀察 Ec 可以影響 Ed。假設 Ec 執行開始的時間較 Ed 小。雖然 Ec 的目的端是 Ne，而 Ed 的目的端為 Nh，Ec 仍有機會去影響 Ed，只要存在一條 Ne 通往 Nh 的路徑。相反，如果並不存在這樣的路徑，Ec 無法影響 Ed 的執行。假設存在這樣的路徑，如前一節所提到的，必需計算經過這條路徑需花費的最小時間，在此稱呼它為 PLAeh。假如 Ec 執行的時間加上 PLAeh> Ed 執行的時間，則 Ec 與 Ed 都是安全事件，相反，只有 Ec 為安全事件。

截至目前為止，我們所提到的 lookahead 指的是封包傳送時所需要耗費的時

間 (鏈結上的延遲+封包流出網卡的時間)，本地運算事件都在同一點上執行並不會使得時間進展。然而，本地計算事件也可能具有非零的 lookahead，這主要和協定模組的設計有關。在這樣的情況下，即使 E1 和 E2 執行在同一點上，如果 E1 被執行的時間加上 lookahead 大於 E2 的時間，那麼 E1 無法影響到 E2。

對於 E1 是本地計算事件，而 E2 是封包傳送事件這樣的案例中，如果 DstNID1 ≠ DstNID2，E1 仍可能影響 E2 的執行。試著觀察圖中 Ea 與 Ec。假設 Ec 執行的時間小於 Ea，Ec 的目的地為 Ne，而 Ea 的目的地為 Nf。如同前面所描述的，假設從 Ne 到 Nf 必需花費的最小時間為 PLAef。如果 Ec 的時間加上 PLAef 大於 Ea 的時間，那麼 Ec 無法影響到 Ea，相反，Ec 將影響 Ea。

對於 E1 是封包傳送事件，而 E2 是本地計算事件這樣的案例中，如果 DstNID1 ≠ DstNID2，E1 仍可能影響 E2 的執行。試著觀察圖中 Ea 與 Ec。假設 Ea 執行的時間小於 Ec。Ec 的目的地為 Ne，而 Ea 的目的地為 Nf。如同前面所描述的，假設從 Nf 到 Ne 必需花費的最小時間為 PLAfe。如果 Ea 的時間加上 PLAfe 大於 Ec 的時間，那麼 Ea 無法影響到 Ec，相反，Ea 將影響 Ec。

3.3 ELP for Wireless Networks

從第三章開始，我們所探討的議題都是從有線網路的觀點出發。在有線網路環境下，每條鏈結 (link)都是雙向的，它包含送和收。在雙向鏈結的條件裡，封包從 Ni 傳給 Nj 與從 Nj 傳給 Ni 彼此是相互獨立的，並不會影響到雙方的傳輸行為。但在無線網路環境下，每一筆封包都是利用廣播的方式傳送，在傳輸範圍下的所有節點都會捕抓到封包送出的訊號，以致於，網路模擬器必需為所有含蓋在傳輸範圍裡的節點排程一筆封包傳送事件。

無線網路並不像有線網路，兩點之間具有一條專屬的鏈結，封包從 Ni 傳給 Nj 與從 Nj 傳給 Ni 會相互影響，因為它們是共用相同的無線通道 (wireless channel)，所以同一個節點無法同時做收、送的動作。

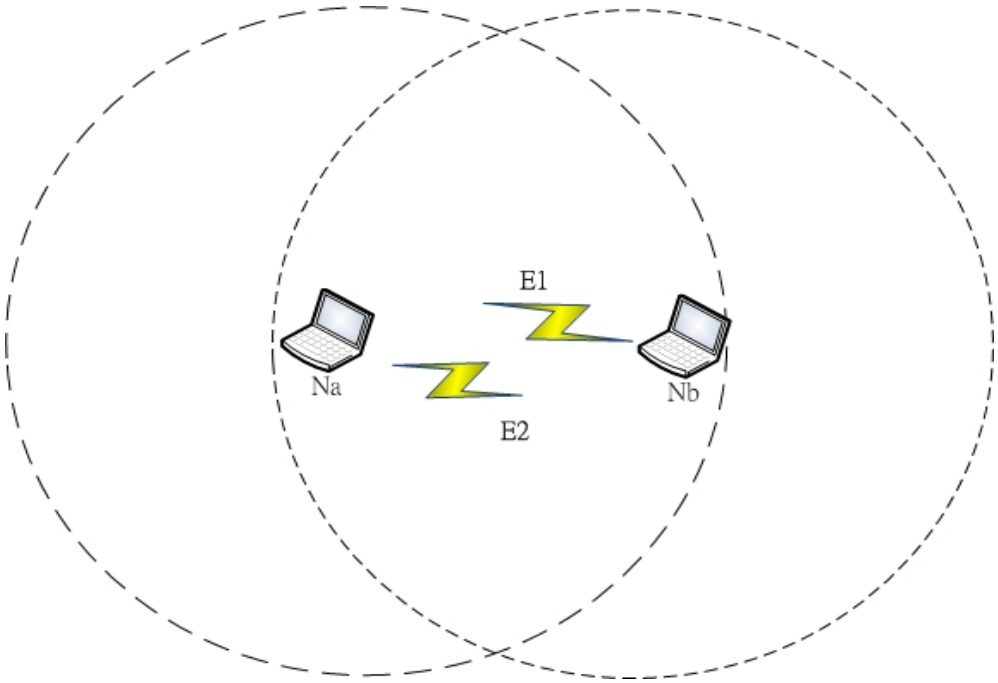


Figure 3.9 The defect of finding safe events rules in wireless network

3.2 節中的條件 1: SrcNID1 ≠ SrcNID2且DstNID1 ≠ DstNID2，在無線網路的環境中，必需增加SrcNID1 ≠ DstNID2與SrcNID2 ≠ DstNID1的條件，才能確保事件是否安全。在此使用一個簡單的例子說明，如 Figure 3.9 所示，試著觀察 E1 是否會影響 E2。假設 E1 與 E2 的執行時間是相同的。當無線節點要傳送封包之前，會先發出 RTS 控制封包告知週遭的節點，它打算傳輸封包，請在這段時間不要跟它競爭通道，收到 RTS 封包的節點會取消原本已經排程要傳輸的封包，等待下次的傳送。因此，如果條件 1 只具備SrcNID1 ≠ DstNID2與SrcNID2 ≠ DstNID1的判定，可能會造成原本應由 Na 傳輸，Nb 等待下一次機會，變成 Nb 傳輸而 Na 等待下一次傳送機會，使得模擬結果大不相同。所以，在無線網路的環境中增加SrcNID1 ≠ DstNID2與SrcNID2 ≠ DstNID1的條件是必要。

為了將事件層平行模擬方法應用在無線網路，像是點對點網路 (ad-hoc network)，鏈結的延遲非常重要，因為 lookahead 的大小將決定 master thread 可以找尋安全事件個數的多寡。因此，在無線網路裡，首先必需決定每一個節點的傳輸範圍內，涵蓋哪些節點，就可以確定 Nodej 是否在 Nodei 的範圍內，如果是，

那就像有一條有線的鏈結從 Nodei 連接到 Nodej。如 Figure 3.10 所示。無線網路的 lookahead 指的是無線鏈結上的延遲 (訊號從 Nodei 傳到 Nodej 所需的傳遞時間)加上封包從網卡出去的傳輸時間。重覆執行上面的敘述，在檢測完所有無線節點的關係後，我們可以將無線網路視為有線網路。所以，有線網路中用來檢測安全事件的條件，在無線網路下仍適用。對於一個會移動的點對點網路 (mobile ad-hoc network)，由於移動的關係，所以網路拓樸無時無刻都在改變，拓樸的改變相對造成點與點之間鏈結延遲的改變，為了確保模擬的正確性，必需要隨時注意點移動的狀況。所以移動點對點網路將不在這裡詳加探討。

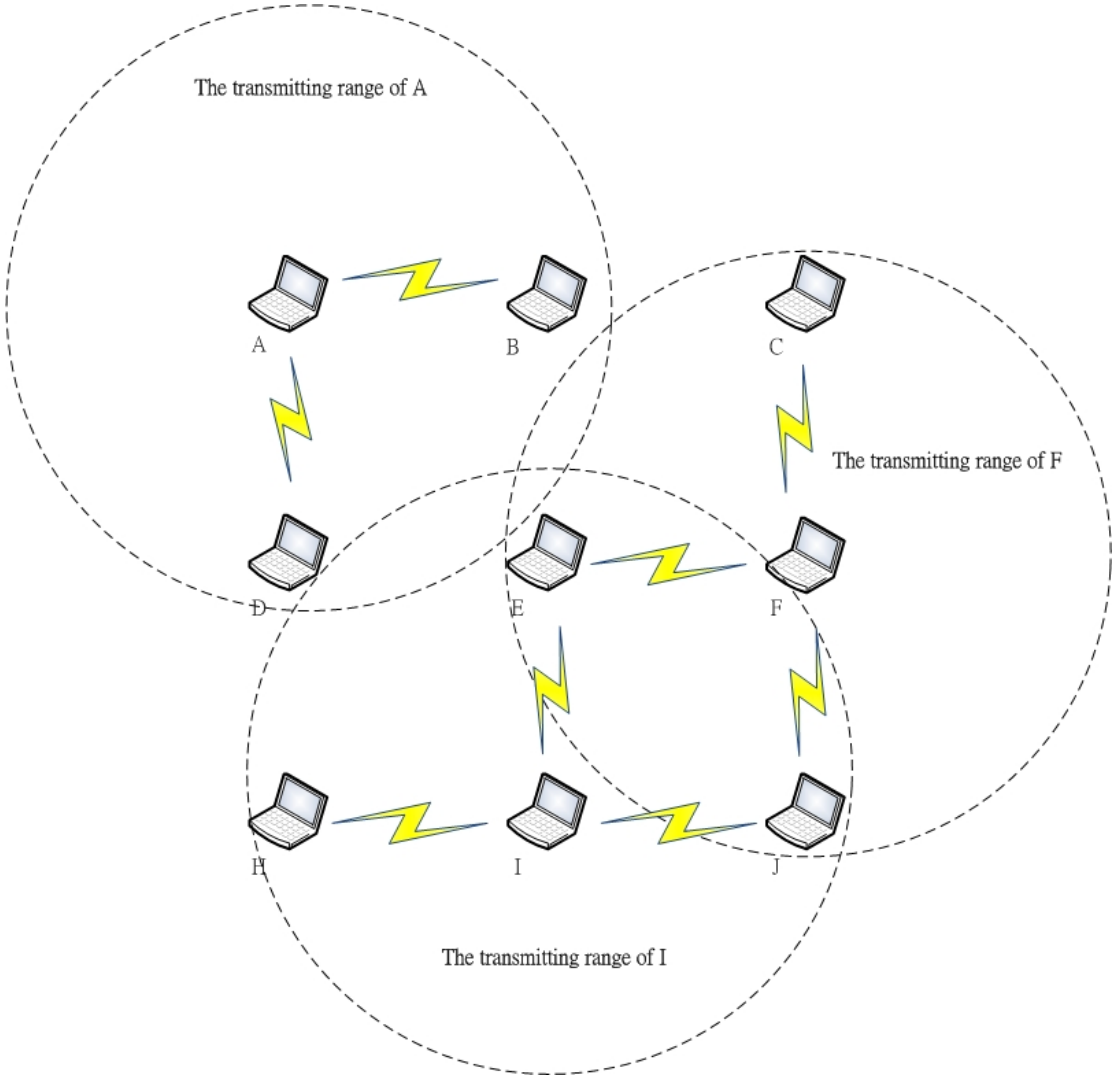


Figure 3.10 Wireless Network Topology

Chapter 4 Application of ELP to NCTUns

在此章節中，說明如何修改 NCTUns 網路模擬器，使得事件層平行模擬方法可以在 NCTUns 架構下正確運行。章節一開始，介紹如何修改 NCTUns 模擬器引擎 (engine)、核心 (kernel)，並提出在修改中所碰到的問題及解法，在下一小節中，更詳細指出 master thread、worker thread 和 thread IPC 的設計及實作。在章節最後，敘述為了能夠在有線及無線網路環境底下使用事件層平行方法做了哪些修改。

4.1 Modifications Made to the NCTUns Simulation Engine

NCTUns 網路模擬器由 C++協定模組與 TCL 檔組成。模擬開始執行時，會先讀入一個網路拓樸的描述檔 (.tcl 結尾檔案)，模擬引擎會根據描述檔給定的參數，創建出 C++協定模組，為了將事件層平行方法應用在 NCTUns 裡，有部份的 TCL 和 C++的檔案必需被修改，接下來，我們將描述 NCTUns 模擬引擎和核心的修改，並提出上小節問題的解法。

4.1.1 Scheduler

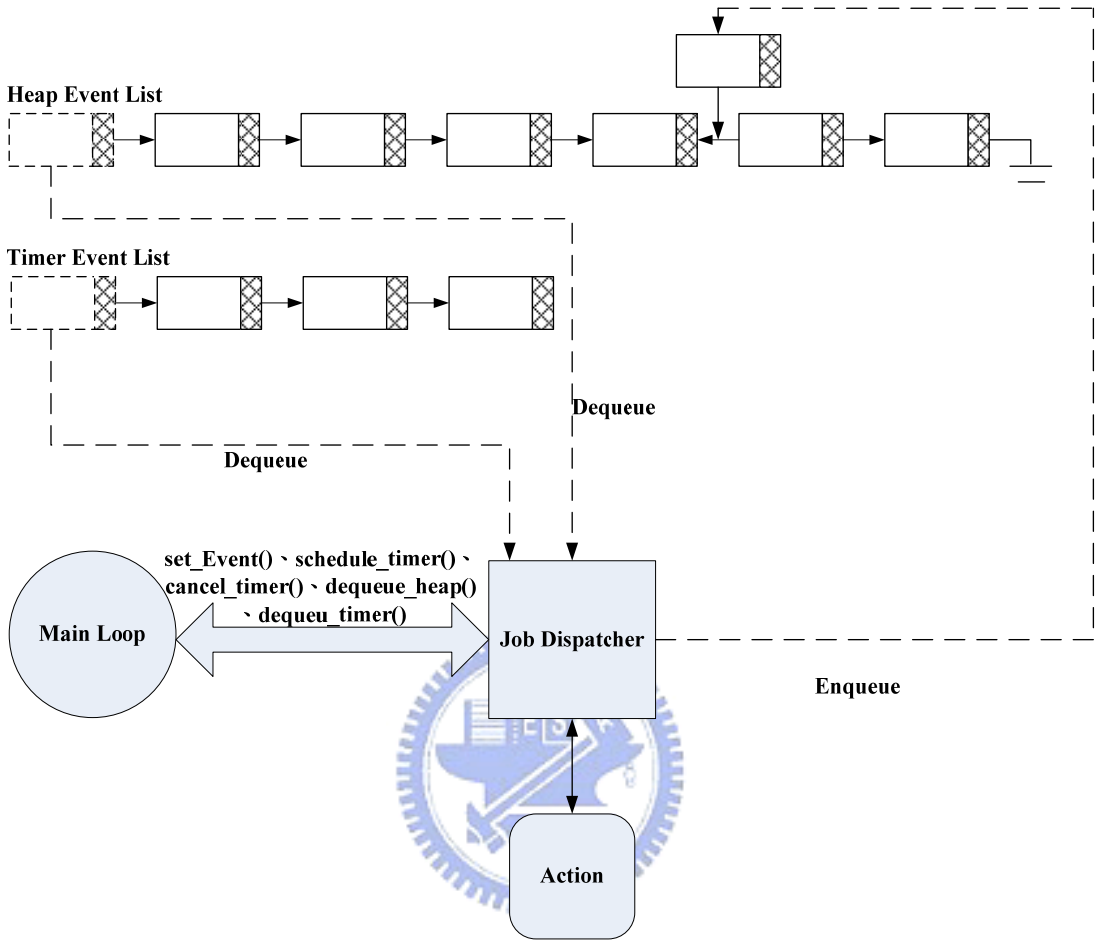


Figure 4.1 NCTUns Scheduler Architecture without ELP

NCTUns 網路模擬器使用 TCL 描述檔去強調網路拓樸的排列方式。在模擬執行開始階段，NCTUns 會先讀進描述網路拓樸的檔案並設定相關的網路環境參數。然後 NCTUns 模擬器會要求排程器 (Scheduler)開始排程事件 (它呼叫 run 函式開始執行網路模擬)。它從事件串列中抓取一個時間單位最小的事件，循序執行。排程器會不斷重覆執行上述的行為直到模擬結束 (當事件串列為空，或是事件時間超過模擬結束時間)。

為了能夠將事件層平行模擬方法應用在 NCTUns 網路模擬器，在開始模擬之前，排程器必需先算好所有點之間的最短路徑，根據 CPU 個數創出相對應的

work threads，並且準備好全域事件串列與安全事件串列。我們將展示這些修改，在之後的小節。

在排程器內會呼叫的函式 set_Event()、schedule_timer()、cancel_timer()、dequeue_heap()、dequeu_timer()，為了使用事件層平行方法，都需要被修改。因為它們都會接觸到全域事件串列和安全事件串列。如 Figure 4.1 描述，在 NCTUns 網路模擬器架構裡，事件串列分成 heap 事件串列與 timer 事件串列，排程器需同時維護這兩類事件串列。排程器也提供許多的 API 去控制它們 - set_Event()、schedule_timer()、cancel_timer()、dequeue_heap()、dequeu_timer()。我們將簡單的介紹這些函式的用途。

- set_Event() 安插一個新的事件 (非時間事件)到 heap 事件串列。
- schedule_timer() 安插一個新的事件 (時間事件)到時間事件串列。
- cancel_timer() 從時間串列中移除一個時間事件。
- dequeue_heap() 從 heap 事件串列中取出一個時間單位最小的事件。
- dequeue_timer() 從時間事件串列中取出一個時間單位最小的事件。

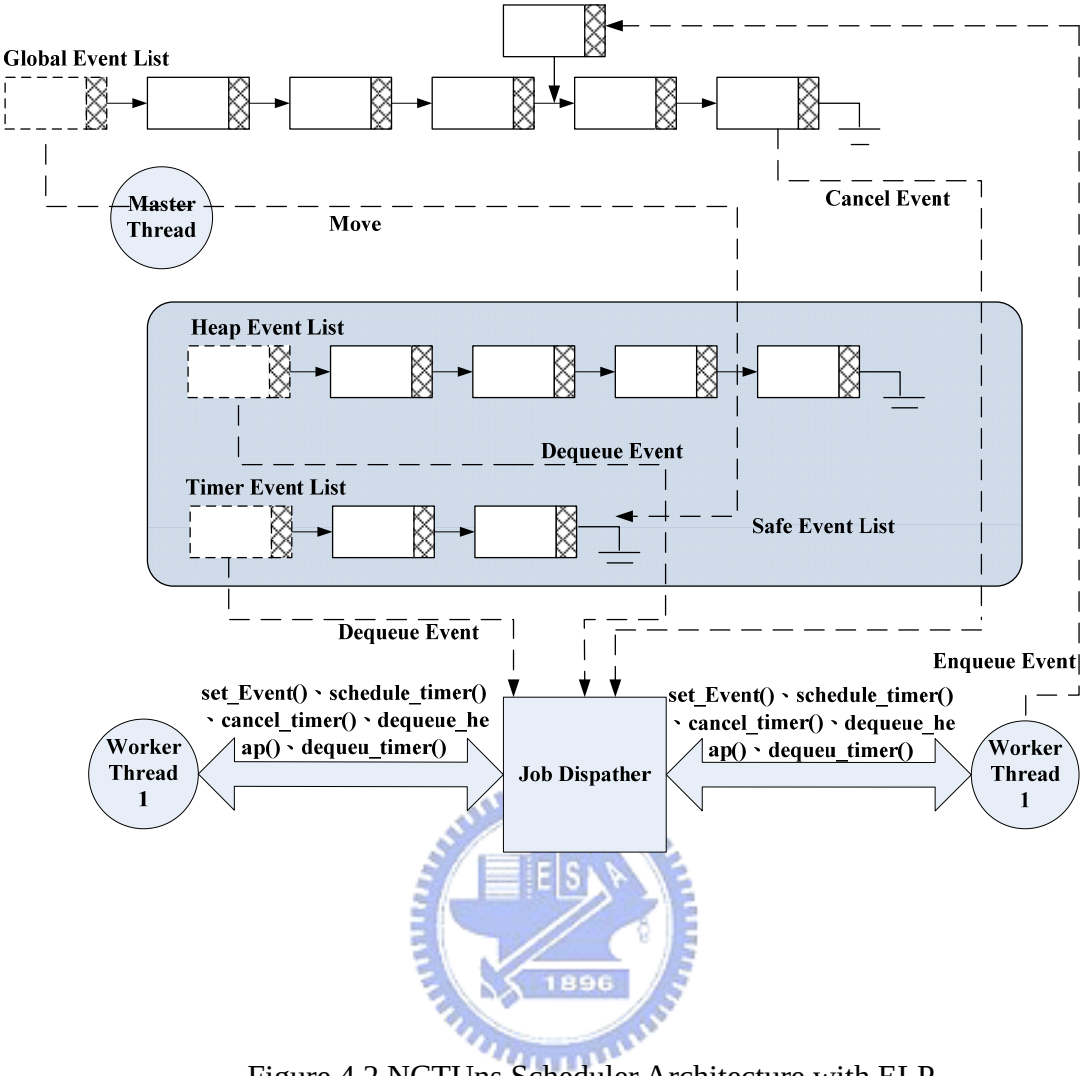


Figure 4.2 NCTUns Scheduler Architecture with ELP

Figure 4.2 展示在事件層平行方法下，排程器新的架構。使用事件層平行方法，事件串列從原有的兩條變成三條，原有的 heap 事件串列與時間事件串列在新的架構中成為存放安全事件的串列，新創造的事件串列，則存放還未被決定為安全事件，及模擬進行中產生的新事件。我們必需修改上述的函式，使得事件層平行方法可以整合進原有的 NCTUns。我們認為最簡單修改函式 dequeue_heap() 和 dequeue_timer()的方法就是將原有的事件串列 (heap and timer)視為安全事件串列。透過這樣的方式，我們不需要去改變原有的架構，因為 worker thread 從安全事件串列中取出事件執行，仍是先執行串列中時間單位最小的事件。不同於 dequeue_heap()與 dequeue_timer()，被修改後的函式 set_Event()和 schedule_timer()，

安插新的事件到全域事件串列,而非安全事件串列 (NCTUns 原有的事件串列)。cancel_timer()的修改不像上述幾個函式這般容易，因為在事件層平行方法的架構下，事件串列不再是單純的儲存未執行事件，而是根據不同種類的事件 (安全事件或非安全事件)，決定應該要把它存入安全事件串列，或是全域事件串列。因此，要取消一個事件必需同時掃描這兩類事件串類。

無論是安全事件串列，或是全域事件串列，都應該受 lock 的保護，這是因為上述在排程器使用的函式 set_Event()、schedule_timer()、cancel_timer()、dequeue_heap()、dequeu_timer()，都有可能同時接觸安全事件與全域事件串列。因此，我們使用 POSIX 執行緒函式庫提供的 API 鎖住可能會發生錯誤的關鍵區 (critical section)。

```
struct ELP_Info
{
        unsigned int src_nid_;
        unsigned int dst_nid_;
        u_int64_t timeStamp_;
        u_int64_t tx_time_;
        Event *owner;
        …
};

typedef struct event
{
        …
        u_int8_t event_type_;
        struct ELP_Info elp_info_;
        …
}Event;
```

Figure 4.3 The ELP_Info Structure

4.1.2 Event Lists

在上一節已經解釋，原本在 NCTUns 中用來儲存未執行事件的串列,在事件層平行方法的架構下，將被視為安全事件串列。我們需要額外創造出一條新的事件串列，用來存放未處理的事件，這些事件並不需要先判斷是否為安全事件，只需按照時間先後做排序。除了新的事件串列外，原有 NCTUns 所使用的事件資料結構 (event data structure)也需要修改。如 Figure 4.3 解釋，我們擴展了事件資料結構，讓事件資料結構可以存放來源點的 ID 及目的點的 ID。這些訊息將被用來決定在全域事件串列中的事件是否為安全事件。為了避免這些欄位是空的，所以在宣告事件結構時,我們使用初始值 65535,如果一個事件帶的來源點 ID 是 65535，master thread 將視這個事件是不安全的。

4.1.3 Simulation Clock

一般循序模擬的模式下，在任何給定的時間去觀察執行中事件的個數，可以發現都僅有一筆。因此，我們只需要一塊空間去存放目前模擬時間，它可以使用函式「GetCurrentTime()」取得。對於事件層平行模擬方法，假設目前在模擬器中有N條worker threads，它們也許在執行的過程中需要取得當前的模擬時間。然而，當前的模擬時間，對於不同的worker thread可能有不同的值，因為模擬時間的進展取決於執行事件所擁有的時間戳記。

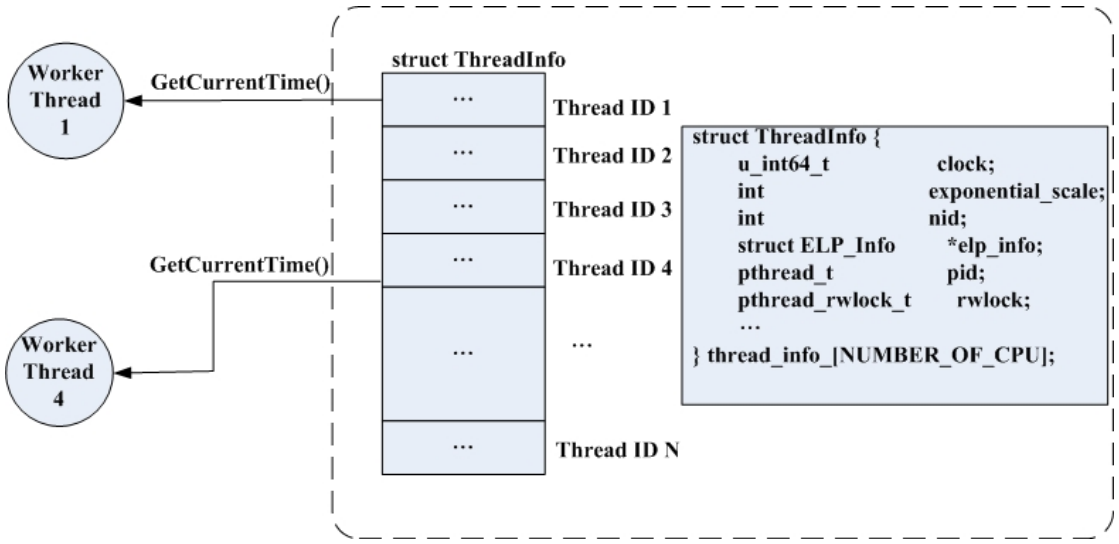


Figure 4.4 Getting Simulation Clock in NCTUns with ELP

我們必需為每一個worker thread準備各自的空間去紀錄當前模擬時間，並且修改原有函式「GetCurrentTime()」的行為，使得它無論被哪一條worker thread呼叫，都可回傳呼叫者所維護的模擬時間。因為上述的理由，我們需要為不同的worker threads準備一塊獨立的空間，程式的修改上使用一個陣列結構完成。每一條worker thread使用自己帶的執行緒ID做為索引去存取這塊空間。如同Figure 4.4所看到的，當一條worker thread在執行安全事件過程中，需要使用到當前的模擬時間時，它可以呼叫函式「GetCurrentTime()」完成這個運算。函式「GetCurrentTime()」會從存放所有worker thread訊息的陣列中以worker thread ID做為索引值，取得模擬時間。

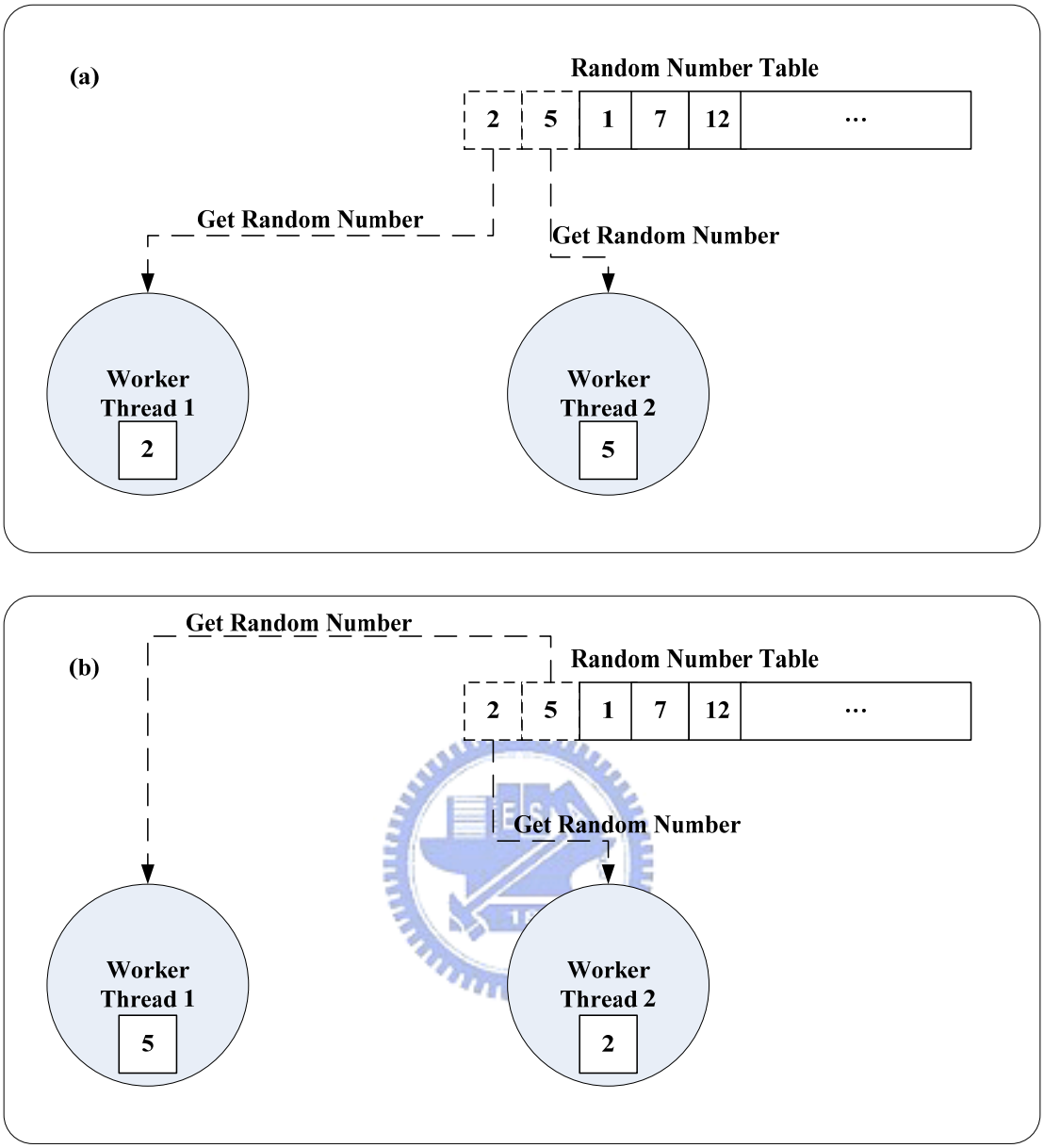


Figure 4.5 Random Number Boundary Problem

4.1.4 Random Number

在 Linux 系統內，一個行程被創建出來，都會被給定一張亂序表 (random table)，行程創出的執行緒，不論是使用者執行緒或是核心執行緒都共享同一張亂序表。然而這樣的狀況會導致模擬結果的錯誤，因為 worker threads 之間在取得亂數的同時，將產生出 race condition 的問題。我們將使用 Figure 4.5 解釋為什麼會產生 race condition。在圖中，有兩條 worker thread，和一張亂序表，亂序表的順序分

別是 2、5、1、7、12、…。假設 worker thread1 和 worker thread2 同時間都有事件在執行，而且它們在事件執行期間都需要從亂序表裡取得亂數。例如，worker thread1 正在執行一筆要計算 random back-off 的事件 E1，它的來源點 ID 是 N1。此時，E1 需要一個亂數決定要在哪一個時間點送出控制或是資料封包。Worker thread2 執行另一筆事件 E2。同樣的，E2 也是一筆計算 random back-off 的事件，不過它的來源點 ID 是 N2。由 Figure 4.5 (a)看到的，同時間各有一個亂數分配給 worker thread1 與 worker thread2。如果 worker thread1 在 worker thread2 之前取得亂數，則分配給 worker thread1 的亂數是 2 而 worker thread 是 5。相反的，如果 worker thread2 在 worker thread1 之前先取到亂數，則它們獲得的亂數將是 5、2，如 Figure 4.5 (b)所示。

上述的例子說明，當 worker threads 取得的亂數不同，所選擇送出封包的時間就不同。然而，如果執行事件的順序每次都不相同，無法保證事件層平行模擬是否有提昇效能的作用。除此之外，Linux 系統中的排程器也是影響事件執行順序的因素之一，因為排程器總是選擇優先權較高的行程執行。不過，當行程的優先權相同時，它選擇可執行串列前端的行程先執行，至於可執行串列的排序，則根據安插到可執行串列的順序決定。

為了解決上面敘述的問題，我們修改原本產生亂數的函式「Random」。這個函式會依據事件結構中儲存的來源點 ID 去計算出一個新的亂數，計算亂數的方式由下列的公式決定:

$$\textbf{Random Number} = (\text{NIDsrc} \times 0\text{x}9\text{e}37001) \bmod 2^{64}$$

使用上述的公式所得的亂數，不會再因為 worker threads 執行順序的不同而有不同的結果。

4.1.5 Global Data Protection

NCTUns 網路模擬器中存在兩個關鍵區 (critical section)，為了把事件層平行模擬

整合至 NCTUns 裡，就得使用 lock 保護。其中一個是封包結構中的 SDATA_Info，另一個則是 log file 的處理。NCTUns 封包結構由 PT_DATA、PT_SDATA 和 PT_INFO 組成，如圖 Figure 4.6 描述。PT_DATA 與 PT_SDATA 被用來存放 user-level 資料。PT_SDATA 如同它的名字，S 代表的是共享 (share)。無線網路的模擬，封包在底層會複制多份給傳輸範圍內的所有點，每筆複制的封包內的 PT_DATA 與 PT_INFO 都會使用新的記憶體空間儲存，但 PT_SDATA 則是所有複制封包共享，有多少封包共享，會使用一個 p_refcnt 計數。當回收封包時，會檢查 p_refcnt，假設有兩個封包共享，那 p_refcnt 的值是 2，每回收一個封包，p_refcnt 就會減一，如果同時間有多個 work threads 要回收封包，它們有可能同時間改到 p_refcnt 的值。因此，更改 PT_SDATA 時，必需使用 lock 去保護，避免未知的記憶體錯誤發生。

另一方面，log file 用來紀錄網路模擬過程中封包傳輸及接收的狀況。因為 NCTUns 網路模擬器，會將整個過程儲存到同一份檔案裡。當超過一個 worker thread 同時想要寫入，就會導致 race condition 發生。在此，我們使用 POSIX 執行緒函式庫提供的 mutex lock 保護。

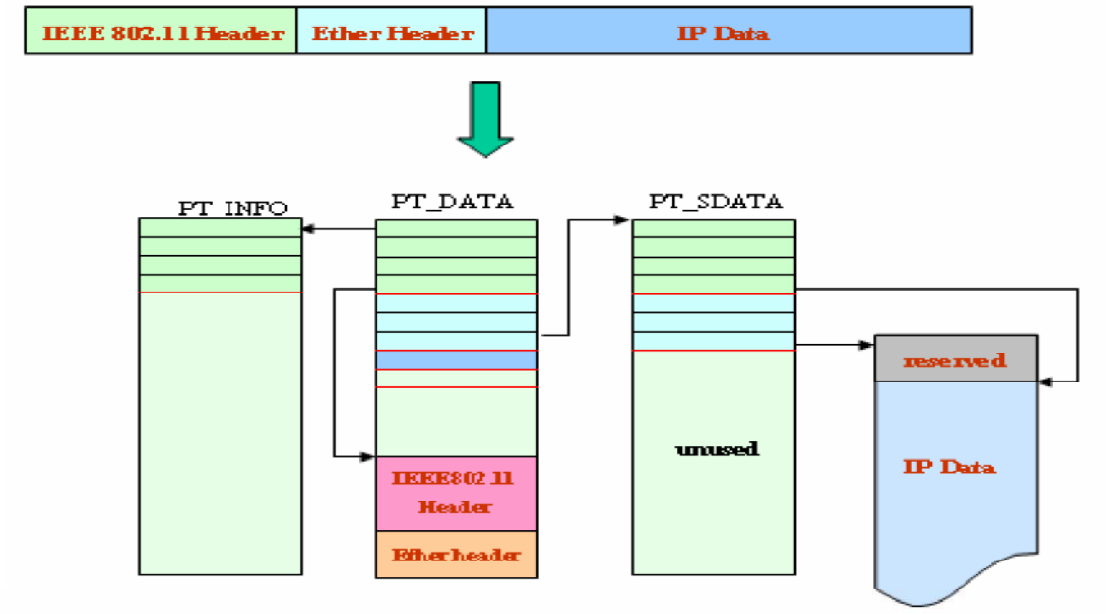


Figure 4.6 Packet Format in NCTUns

4.2 Modifications Made to the NCTUns Simulation Kernel

接下來這個章節中要介紹的是使用事件層平行模擬，對於 NCTUns 核心 (kernel) 所做的修改。4.1 節提出的問題也會在此小節中提出解答。

4.2.1 Task Structure & System Call

行程描述器紀錄所有與行程相關的資訊，包含它使用的記憶體區段，識別碼 (PID)，父行程的資訊，等…。NCTUns 在單核心系統的架構中，為了能夠識別由 NCTUns 模擬引擎創建出來的行程，在行程描述器中加入新的欄位 (nctuns_task_struct)，如 Figure 4.7 (a)所示。nodeID 紀錄新行程在 NCTUns 的哪一個節點上被執行，透過檢查 nodeID，可以馬上得知目前執行中的行程是否為 NCTUns 創建出來的行程，如果是，nodeID 不為零，反之，nodeID 為零。NCTUns 核心也提供許多函式可以快速檢測執行中的行程是否屬於 NCTUns。

```
struct task_struct
{
        …
        struct nctuns_task_struct;
        …
};
```

(a)

```
struct nctuns_task_struct
{
        unsigned long nodeID;
        unsigned long endtime;

}nctuns_task;
```

(b)

```
struct nctuns_task_struct
{
        unsigned long nodeID;
        unsigned long endtime;
        int     state;
        u_int64_t curr_time;
        struct task_struct *wakeup_proc;
}nctuns_task;
```

Figure 4.7 The nctuns_task_struct Structure

將多核心的系統整合至 NCTUns 的架構中，最大的問題是模擬時間的處理。從 4.1 和 4.2 節可以知道，使用事件層平行模擬方法，模擬引擎會在程式執行開始時創建許多 worker threads，同一時間內，可能會有許多的 worker thread 從安全事件串列中取出不同的安全事件執行，由 4.2.3 節得知，worker threads 將各自維護的時間存放在一個以執行緒 ID 為索引的陣列中。模擬引擎在 user-level 可透過 API「GetCurrentTime()」存取當前時間資訊，但在 kernel-level 無法得知正確的模擬時間。因此，我們再一次的修改行程描述器的架構，在原有的資料結構中，增加「curr_time」和「wake_up_proc」欄位，如 Figure 4.7 (b)所示，「curr_time」紀錄當前模擬時間而「wake_up_proc」紀錄目前的行程是被哪一條 worker thread 給喚醒，並增加新的系統呼叫，使得 worker thread 可以將模擬的時間寫到行程描述器裡的「curr_time」欄位。透過修改行程描述器，讓每一條 NCTUns 的執行緒或是應用程式都擁有自己的模擬時間。

4.2.2 Global Data Protection

NCTUns 的核心，存在兩個重要的資料結構，分別是 tunctl_ec 和 callwheel。tunctl_ec 與一般 tunnel interface 在結構上是相同的，但除了 tunctl_ec 外，所有 NCTUns 產生的 tunnel 用途上被視為如真實世界的網路界面卡，主要負責把上層的封包推到底層鏈結上。tunctl_ec 在 NCTUns 的架構中主要負責通知模擬引擎，核心有新的事件想要加入排程。所有由核心產生的事件必需經由 NCTUns 排程器的篩選後，才可以被執行。因此，只要 kernel-level 有新的事件產生，無論是有新的封包要送到下層的協定模組或是屬於 NCTUns 的應用程式排定一個計時器，它都必需先使用 tunctl_ec 告知排程器，請排程器把新的事件放到事件串列。

由於事件層平行模擬在多核心的系統中運行，所以同一時間內可能會有多個應用程式、或是 worker threads 同時存取 tunctl_ec 傳遞的訊息，因此必需使用 lock 來鎖住 tunctl_ec，避免造成多個行程存取，導致資料的破壞。

另一個重要的資料結構存放的是 NCTUns 在核心內的時間計時器串列。核心計時器 (kernel timer)是除了中斷處理、系統呼叫外，另一種通知核心的方法。計時器最大的好處就是可以指定核心在某個時間點，去完成某些事。NCTUns 網路模擬器使用的時間並非真實世界時間，所以，無法將 NCTUns 在核心中的計時器掛在原有的核心計時器串列中，這可能會使得 NCTUns 計時器起來的時間太快或太慢。因此，NCTUns 必需在核心裡創造一條不屬於 Linux 核心管理的時間計時器串列。

callwheel 是 NCTUns 核心用來管理計時器創出的一條串列，在這條串列裡的所有計時器所持的時間都是 NCTUns 看到的虛擬時間。對於原有單核心系統的 NCTUns 而言，callwheel 並沒有問題，但在多核心系統，由於同時間可能會有多支行程或執行緒存取它，那麼就極有可能造成串列中資料的破壞。使用事件層平行模擬方法時，callwheel 的存取，我們也必需使用 lock 去限制同一時間至多只有一個行程或執行緒可以去更動內部的資料。

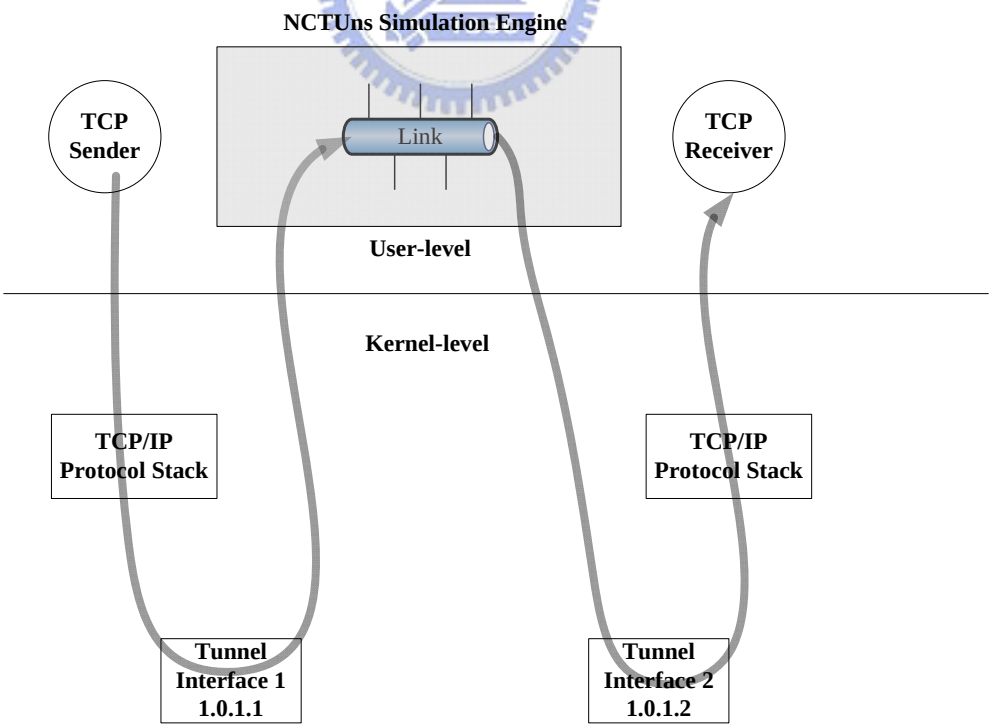


Figure 4.8 TCP Connection Scenario in NCTUns

4.2.3 Encountered Problems

在 2.4 節簡單介紹了 NCTUns 網路模擬器架構，這一小節中將列出事件層平行模擬方法應用在 NCTUns 上，會遭遇到的問題。在此使用一個簡單的例子說明，如 Figure 4.8 所示，圖中模擬的是一個點對點 TCP 傳輸。在 NCTUns 的架構下，所有真實世界傳輸封包的軟體，都不需要修改，就可以在 NCTUns 網路模擬器裡執行，最主要的原因是 NCTUns 內的 TCP/IP 協定模組是採用 Linux kernel 內真實世界流量所會執行的TCP/IP協定模組，而非使用者層面 (user-level)的程式。所以，真實世界所使用的流量產生器，跑在 NCTUns 核心內的行為與真實世界並無差別，只是它的時間不在依循真實世界的時間，而是 NCTUns 提供的虛擬時間。因此，每一支在 NCTUns 上執行的應用程式，就像執行在真實世界的某一台電腦上，每支程式在核心中都擁有自己的行程描述器，並且它們都會與 NCTUns 模擬引擎在核心排程器內競爭 CPU 時間。

模擬時間在 NCTUns 架構中，是採用事件驅動跳時的方式進展。簡單的說，模擬引擎在開始執行每筆事件時，會將模擬時間更改為事件所帶的時間。由於模擬引擎與其它 NCTUns 產生出來的流量產生器在核心中都是確實存在的行程，因此它們都會被排進核心排程器的執行佇列中，與其它的應用程式競爭 CPU 時間，所以並不保證，每次執行的結果是相同的。假設目前有三支行程，分別是模擬引擎 (SE)、app1 (TCP sender)與 app2 (TCP receiver)。在時間 T1，SE 執行一筆要叫醒 app1 的事件，如果 SE 完成叫醒 app1 的工作後，並沒有釋放掉 CPU，或是核心分配給它的 CPU 時間還未終結，而繼續執行，模擬時間可能會因為 SE 從事件串列中取出下一個事件執行使得它進展至 T2。在 T2 的時間下，app1 才搶到 CPU 時間，那由 app1 產生的事件將被給定 T2 時間戳記; 另一方面，如果 SE 叫醒 app1 後，馬上釋放 CPU 時間，並且 app1 搶到 CPU，產生新的事件，此時它所擁有的時間戳記將是 T1，不再是 T2。為了解決這個問題，原執行在單核

心系統下的 NCTUns 網路模擬器改進的方法是提高應用程式的行程優先權，所以在 SE 叫醒 app1 後，無論 SE 擁有的 CPU 時間是否已經用完或是 SE 主動歸還剩餘的時間，CPU 使用權將會被較高優先權的 app1 給先佔。因此，每次模擬的結果都會是相同。

上述的解法，只適用於單核心系統。在多核心系統下，無法保證 app1 一定會強佔正在執行 SE 的 CPU，因此沒辦法確定模擬時間不會在 SE 叫醒 app1 後再一次的跳動。這個問題是事件層平行模擬第一個遇到的問題。

單核心系統內，模擬器同時間只會執行一筆事件，至多只有一個行程可以更改模擬時間，所以無論是 user-level 或 kernel-level 所看到的模擬時間都是一致的，但是，使用多顆核心的事件層平行模擬方法，不同於原有的設計架構，可能同一時間點上有許多顆不同的核心從事件串列中取出事件執行，使得每條正在執行事件的執行緒都有權力可以更動模擬時間，如果存放模擬時鐘的空間仍只有一個，可能會使得時間的跳動變成 T1->T3->T2 (假設 T1<T2<T3)。因此，在事件層平行模擬方法，採用每一條 worker thread 都具有屬於自己的虛擬時鐘，不過這衍生出第二個問題，由於每一條 worker thread 都擁有自己的時鐘，在整個 NCTUns 網路模擬器環境裡，對於從核心產生的事件，就無法得知該使用哪一條 worker thread 所持有的時間，這對於 NCTUns 是相當嚴重的錯誤。舉例來說，在 NCTUns 網路模擬器裡，每一個跑在模擬器上的 socket 所使用的時間都是虛擬時間，多核系統中，由於無法確定該使用哪一條 worker thread 持有的時間，這對於 TCP 的應用程式，會產生極嚴重的錯誤，因為這有可能會造成 TCP 不斷的重傳，或是根本無法完成三向交握 (three-way-handshake)。

4.2.4 The Solutions of Problems

在 4.2.3 節中提到當事件層平行模擬方法整合到 NCTUns 裡會發生的兩個問題，分別是:

- 在多核心系統，無法保證應用程式在執行時，模擬引擎並沒有同步執行，使得模擬時間持繼的進展，造成模擬結果的錯誤。
- 事件層平行方法下，每條 worker threads 都有自己的虛擬時鐘，由應用程式送出的封包，會先使用 4.3.2 節提到的 tunctl_ec 去通知模擬引擎有新的封包要從 kernel-level 進入到 user-level，worker threads 會使用函式「readt0e」從 tunctl_ec interface 接收核心傳遞的事件，如果新事件代表的是封包傳輸，worker threads 會根據自己的虛擬時間，為事件填上開始執行的時間。這樣的狀況會導致於從 tunctl_ec interface 接收核心傳遞事件的 worker thread 不同，新事件未來執行的時間也將不一樣。

4.3.1 節中介紹在行程描述器中加入新的欄位「curr_time」去保存每一支行程的模擬時間，這使得模擬引擎與應用程式都保有自己的時間，即使模擬引擎與應用程式同步執行，也能保證應用程式的時間能按照自我的行為在跳動，這種作法可以解決問題 2 所造成的錯誤。問題 2 中，傳輸封包的時間會由讀取 tunctl_ec 的 worker thread 決定，不過，新的方法令應用程式本身就可以填入封包傳輸事件該發生的時間，所以，不同的 worker thread 去讀取 tunctl_ec 也不會發生模擬的錯誤。可是，這個條件對於解決問題 1 仍就是不夠的。接著，使用一個簡單的例子，如 Figure 4.9 描述，突顯使用新的架構下，會出現的問題。

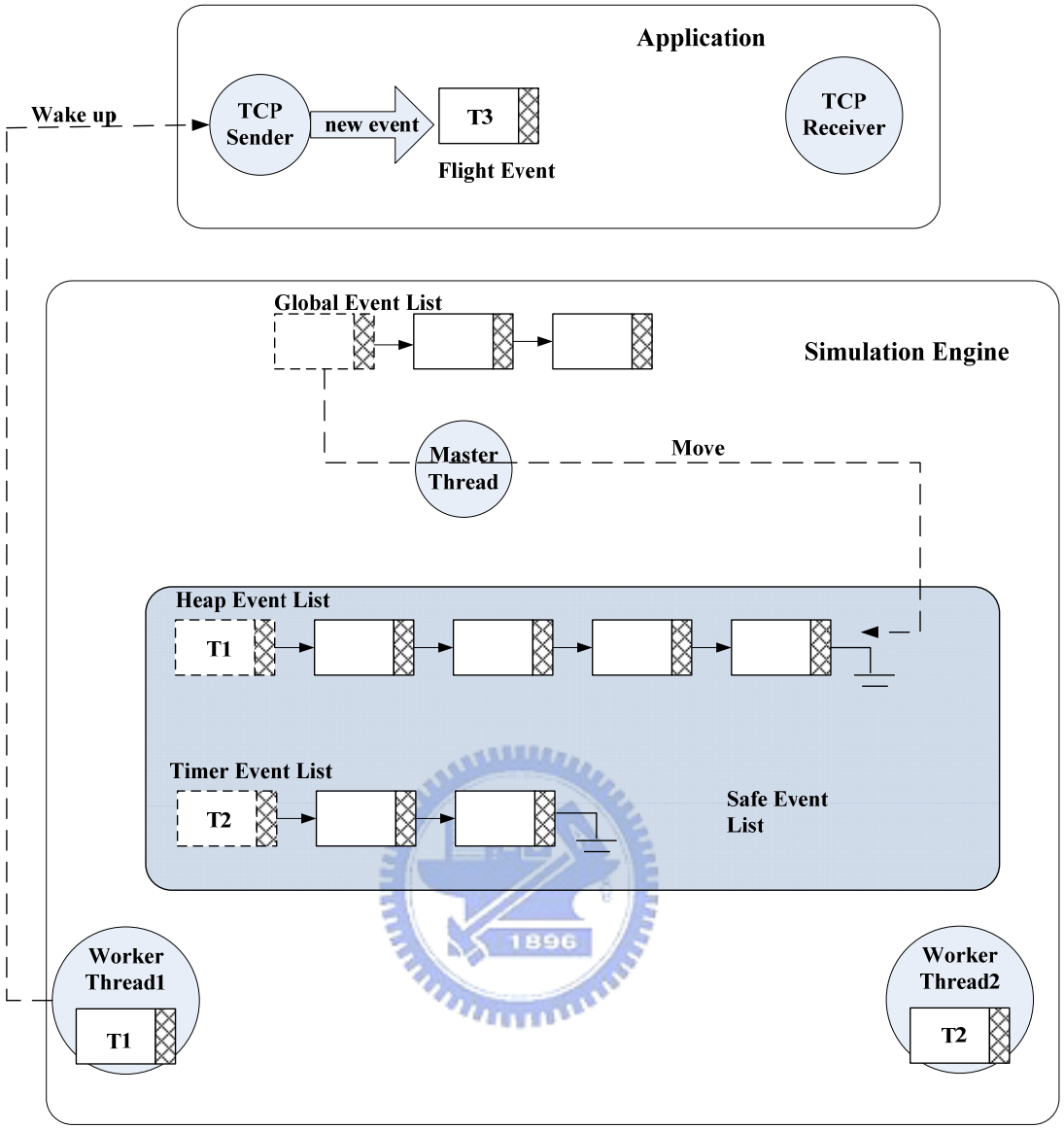


Figure 4.9 Simulation Clock Problem

圖中，存在三條 threads，分別為 worker thread1、worker thread2、master thread，及兩支應用程式 TCP Sender、TCP Receiver。worker thread1 目前紀錄的時間為 T1，worker thread2 目前紀錄的時間為 T2。worker thread1 在叫醒 TCP Sender 後，將繼續從安全事件串列中選取另一個安全事件執行，如果目前並無任何安全事件在安全事件串列中，則 mater thread 會被 worker thread 叫醒，並開始找尋安全事件，此時 TCP Sender 仍在運算，但是由 TCP Sender 送出的封包在所有的事件串列中都不存在，這樣的事件在模擬術語中稱為「flight event」。Master thread 在找

尋安全事件的過程中，不會發現有這樣的事件存在，使得安全事件的檢測上出現漏洞，進而造成模擬錯誤。

由此可見，雖然在行程描述器中紀錄了各個行程的模擬時間，應用程式可以不受到 worker threads 的影響，不過，應用程式和 worker threads 的同步執行，還是衍生出新的問題。為了解決「**flight event**」的發生，應用程式執行時，worker threads 必需停止執行，直到應用程式執行結束後，worker threads 才能繼續執行。相同的例子，當 worker thread1 叫醒 TCP Sender 後，worker thread1 需要拉起重新排程的旗幟，並從核心排程器的執行佇列中移出，放到等待佇列 (wait_queue)。TCP Sender 完成封包傳送的工作後，主動叫醒沉睡中的 worker threads，通知它從 tunctl_ec 裡將新事件搬移至全域事件串列。

worker thread 再叫醒應用程式的同時，會對應用程式的行程描述器資料結構裡寫入關於把它叫醒的 worker thread 的資訊，包含 worker thread 的模擬時間、worker thread 的 PID。應用程式結束工作後，會根據行程描述器紀錄的 PID，叫醒正在等待中的 worker thread。

4.3 The Components of the ELP Architecture

除了修改 NCTUns 模擬引擎和核心外，整合事件層平行模擬方法上還有許多新的元件需要被實作。此小節中，我們將呈現 worker thread、master thread、和執行緒間 IPC 設計及實作方法。此節的最後，呈現如何事先計算點與點之間的最短路徑。

4.3.1 Master Thread Design and Implementation

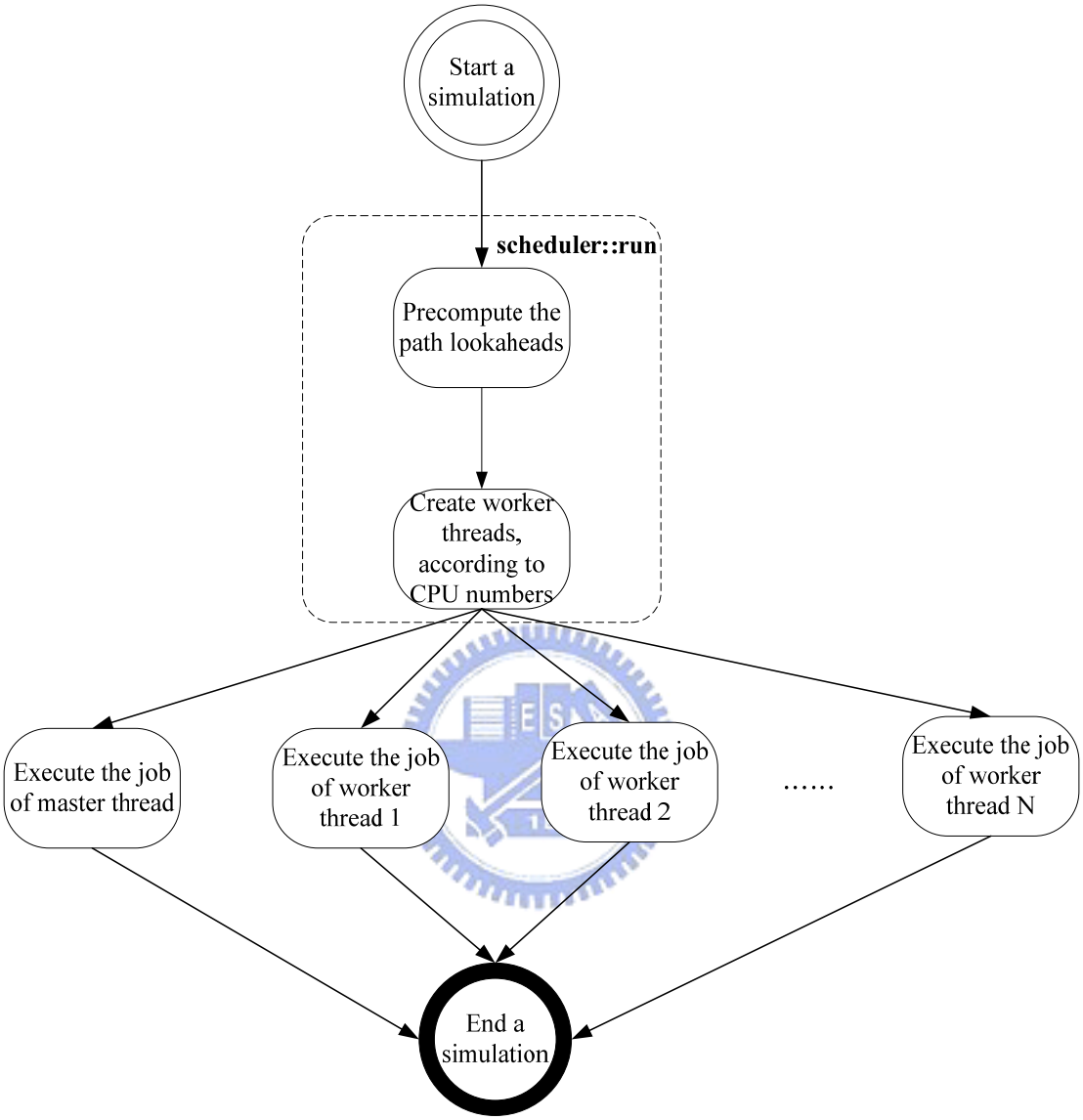


Figure 4.10 Simulation Engine Execution Flow Chart in Using ELP

master thread 主要的任務是從全域事件串列中尋找安全事件，並把安全事件從全域事件串列搬移到安全事件串列。如圖 Figure 4.10 解釋，程式開始執行時，會先呼叫函式 run()，完成點對點之間最短路徑運算，並根據 CPU 個數創建 worker threads。完成上述步驟後，原有程式則開始進入找尋安全事件的 main loop。master thread 利用 3.2 節提到的四組條件開始重覆執行從全域事件串列裡搜尋安全事件

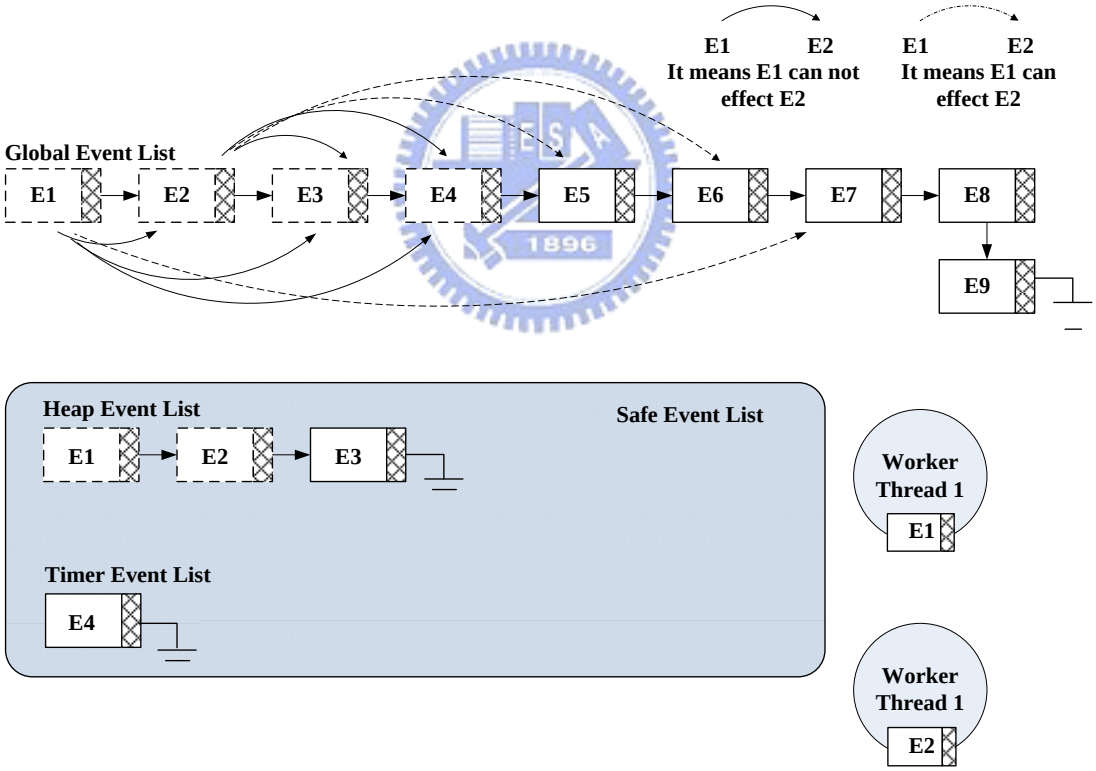
的任務。當 master thread 因為找不到安全事件進入睡眠的期間，只有 worker threads 可以中斷它的休眠，它無法將自己喚醒，這是因為它在事件層平行架構裡是一個被動的角色。

在搜尋安全事件的過程中，我們設置一個變數 MP (maximum Parallelism)，它的值是 worker threads 數目十倍大，ex: 在四核心的系統下，它的值是 30。當 mater thread 找尋到安全事件的個數超過或是等於 MP 時，master thread 將停止繼續尋找安全事件，並進入休眠狀態，等待其它 worker threads 再一次的喚醒它。另一方面，如果 master thread 無法找到與 MP 相等的安全事件數時，它會嘗試至少找到和 worker threads 個數一樣多的事件。然而，為了找尋與 worker threads 數目一樣多的事件，有可能 master thread 要去檢查超過 worker threads 數目的事件或是它此次的搜尋將無功而返。因此，我們設計了一個計數器去存放一個上限值，當搜尋的次數達到上限值，它將停止尋找安全事件。在我們的設計上，這個上限值的大小與 MP 相同。上述的設計，可以有效降底找尋安全事件浪費的 CPU time，因為每執行一次這樣的運算就必需花費 $O(MP^2)$的時間複雜度。

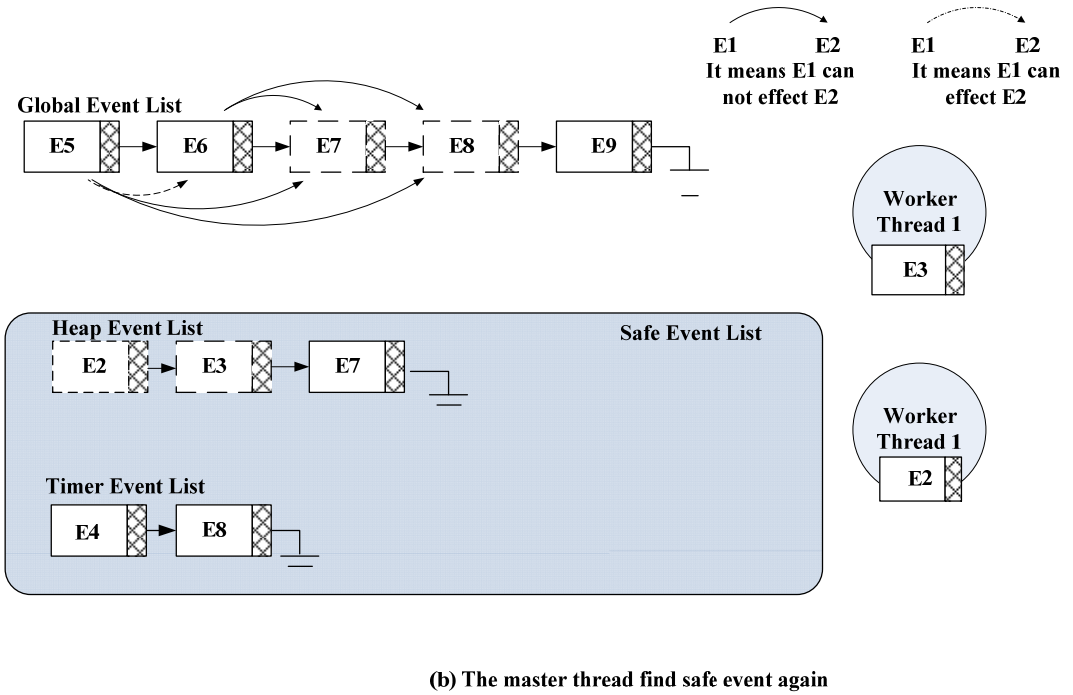
我們給定一個例子在 Figure 4.11 並把 MP 的值設成 20。一開始，假設目前有九個事件在全域事件串列裡，且安全事件串列是空的。如圖中所看到的，為了方便討論，我們替每一個在全域事件串列的事件給定一個名字，從串列的第一個元素開始，依序為 E1，E2，E3，…，E9。圖中從事件 Ei 到事件 Ej 的實心箭頭符號，代表的是 Ei 不會影響到 Ej。另一方面，假如從 Ei 到 Ej 存在一條中空箭頭符號，代表的是 Ei 可以影響 Ej。當 master thread 剛開始找尋安全事件時，它只能從全域事件的頭端找到四個安全事件，分別為 E1、E2、E3、E4，如 Fig. 4.x 所示。這是因為，E2 會影響 E5、E6，E1 會影響 E7、E8、E9，使得 master thread 無法更進一步的找到更多的安全事件。然而，它也已經找出四個安全事件，超過 worker thread 個數，它將安全事件從全域事件串列搬到安全事件串列後就停止安全事件的搜尋。在 Figure 4.11(a)裡，我們在全域事件串列中使用虛線描繪 E1、

E2、E3、E4，這代表它們已經被搬到安全事件串列。我們在安全事件串列中使用虛線描繪 E1 和 E2 代表它們目前正被某些 worker threads 取出或是正在執行。

當 master thread 再一次被 worker thread 喚醒，它將再一次進入找尋安全事件的運算。由 Figure 4.11 (a)和 Figure 4.11 (b)展示，我們可以清楚觀察到 E2 會影響 E5，且 E2 在目前的時間點上仍在 worker thread2 上執行。因此，master thread 無法決定 E5 是否是一個安全事件，而且 E5 也可以影響 E6。假設 E2、E3、E4 不會影響到 E7、E8，master thread 可以把它們從全域事件串列搬移到安全事件串列，並且停止安全事件的搜尋。這是因為 E5 和 E6 都是不安全的事件。這樣的狀況下，master thread 僅僅只能找到兩個安全事件，但是它能提供足夠的事件數給 worker threads 去執行。



(a) The master thread first find safe event



(b) The master thread find safe event again

Figure 4.11 Worker threads perform two times finding-safe-event procedure

4.3.2 Worker Thread Design and Implementation

在事件層平行模擬方法架構下，worker thread 扮演主動的角色。當一個 worker thread 被創建出來，它將試著從安全事件串列中找尋一筆安全事件執行。假如安全事件串列中並不存在任何的安全事件，worker thread 將進入休眠狀態並通知 master thread 開始執行安全事件尋找的運算。為了避免 worker thread 閒置太久，影響效能提升，它本身可以經由計時器的到時被喚醒，。如 Fig. 3.3 所示，當 worker thread 因為計時器到時被喚醒時，仍沒有安全事件在安全事件串列裡，則 worker thread 將再一次進入休眠的狀態，並且計時器的時間將以 exponential 的速度往上增加。如果，worker thread 被喚醒時，可以從安全事件串列中找到安全事件執行，計時器的時間將回復為初始值。

然而，如果每次 worker threads 都是等到安全事件串列都為空時才通知 master thread 去找尋安全事件，這會使得 master thread 在找尋安全事件，worker thread 無法同時執行，浪費了作用在 worker thread 上的 CPU 時間。因此，我們提出一

種方法去解決這樣的問題。當 worker thread 從安全事件串列中取出一個安全事件，它會去檢查目前安全事件串列的長度是否小於一個門檻值，這個門檻值代表的是安全事件串列最小需維持的安全事件數目。在我們的設計上，是以 worker thread 的數目加 1 而此門檻值。所以在四核心系統下，這個值就為 4。因此，當 worker thread 檢查到安全事件串列的長度小於這個值時，它就會要求 master thread 再次尋找安全事件，而 worker thread 也因為安全事件串列還存在安全事件可以執行，避免浪費 CPU 時間。

4.3.3 Threads IPC

在前面的章節中，我們已經介紹了大部份事件層平行模擬的方法。從前幾章的敘述中可以發現，master thread 與 worker threads 間存在一種溝通的方式，使得 master thread 可以通知 worker thread 去安全事件串列抓取一筆事件執行，而 worker thread 也能喚醒 master thread，要求它繼續尋找更多的安全事件。

POSIX 執行緒函式庫提供許多好用的函式給開發者，其中存在兩種函式可以完成上面敘述的功能，分別為 pthread_cond_signal()與 pthread_cond_broadcast()。pthread_cond_signal()僅會通知等待相同條件而處於休眠中的執行緒群組中的一條執行緒，pthread_cond_broadcast()會以廣播的方式通知所有處在相同等待條件群組中的所有執行緒。

當 master thread 想要通知 worker threads 去執行安全事件串列中的事件，它可以使用 pthread_cond_broadcast()喚醒所有休眠中的 worker threads。但是這存在一個問題，假設目前安全事件串列中的事件不足 worker thread 個數，則被叫醒的 worker threads 很可能因為找不到安全事件可以執行而馬上進入休眠狀態，這會使得 CPU 時間被浪費。

因此，當安全事件串列的長度小於 worker thread 個數時，master thread 將不使用 pthread_cond_broadcast()通知 worker thread，而是使用 pthread_cond_signal()

去喚醒與安全事件串列長度相等的個數。

另一方面，worker threads 也會通知 master thread 去尋找更多的安全事件提供給它們執行，但是它們總是使用 pthread_cond_signal()去喚醒 master thread，這是因為整個模擬的過程至多只會有一個 master thread 存在。

4.3.4 Precomputing the Minimum Path Lookahead

在第三章曾經提過，master thread 能找到多少安全事件與 lookahead 的大小有密切的關係。因此，我們必需事先去計算封包在點與點之間傳送需要花費最小時間總和。為了計算所有點的最短路徑，在此我們使用 Floyd-Warshall 演算法事先運算並將結果存放到一個二維陣列，我們稱此陣列為最短路徑表 (shortest path table)。計算封包從一個點傳到另一個點所需花費的最短時間，它需要 $O(N^2)$的時間複雜度，空間上為了存放這些結果，也需要 $O(N^2)$的記憶體空間。Master thread 可以很容易取得陣列中任一元素的資訊，因為它是使用來源點 ID 與目的點 ID 做為索引。

然而，在計算所有點的最短路徑前，我們必需先取得相鄰點之間鏈結上的延遲。使用一個二維的陣列存放相鄰點之間鏈結上的延遲後，利用 Floyd-Warshall 演算法去計算任兩點間的最短路徑。

在 NCTUns 網路模擬器的有線網路中，兩點之間如果相連的話，會存在一條鏈結，鏈結上的延遲會紀錄在所要模擬的描述檔裡。在讀入網路拓樸描述檔後，模擬器會根據讀進來的資料，寫到二維陣列中。完成後，透過 Floyd-Warshall 演算法就可以得到最短路徑表。

不同於有線網路的設計架構，無線網路鏈結上的延遲並不會紀錄在描述檔中。如同在 3.2 節所提到的，無線網路傳送一個封包需要花費的時間，是點與點之間傳送的時間加上封包從網卡傳輸出去的時間，因此，為了計算傳送時間，它需要事先取得所有點的位置，才可以去計算點與點之間的最短路徑。

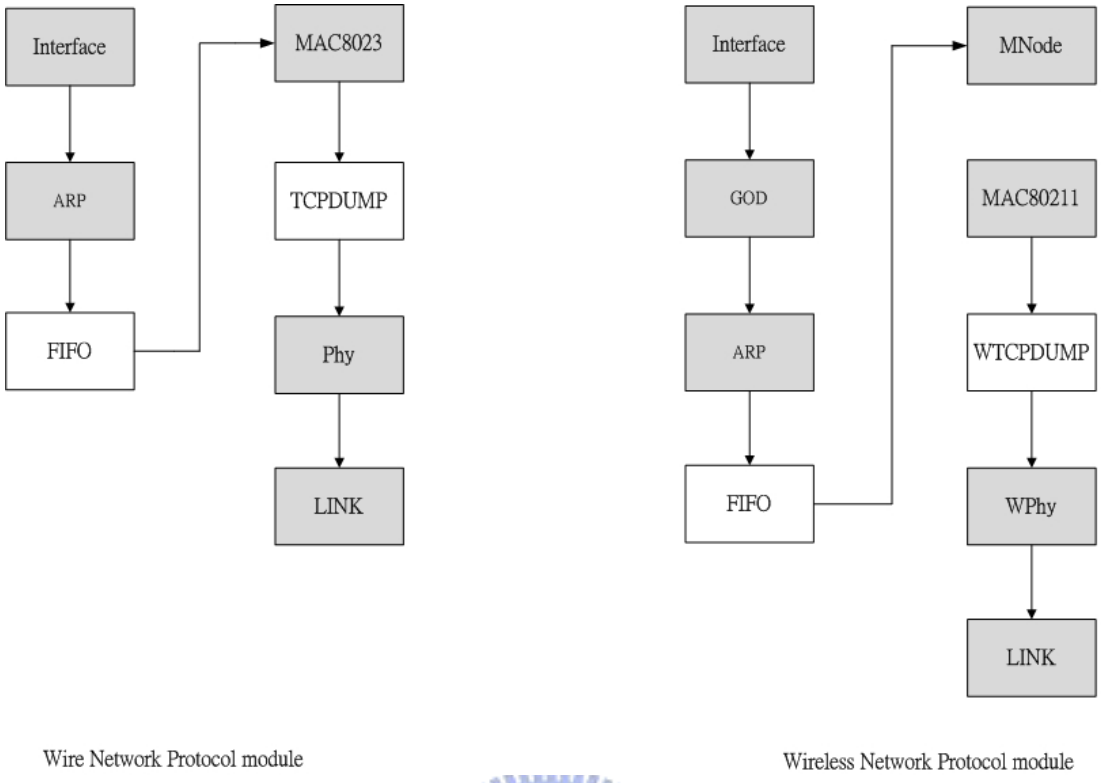


Figure 4.12 Wire and Wireless Network Protocol module

4.4 Modifications Made to the NCTUns Network Protocol Modules

在這篇論文中，我們著重的重點在於 NCTUns 網路模擬器內有線網路及無線網路探討。有線網路及無線網路協定模組如 Figure 4.12 所示。為了配合事件層平行模擬方法，而需要修改的協定模組，在圖中以淺灰色的顏色表示，未被修改的模組則以白色代表。在接下來的章節中，我們將簡短介紹所做的修改。

4.4.1 Wired Network Protocol Modules

為了使用事件層平行模擬方法，給定事件來源點 ID、目的點 ID 及取得有線鏈結上的延遲是很重要的。因此，我們修改的重點在於模組內是否呼叫過排程函式，例如: set_Event()。下面將對修改的模組給定簡單的介紹。

- **Interface**

這個模組是由 NCTUns 網路模擬器，封包從核心模組轉為執行 user-level 模組的進入點，主要接收或是傳送由核心產生的封包。

- **ARP**

  這個模組執行 ARP 機制，而且它也會排定一些本地計算事件 (查尋目的點的 MAC address)。

- **MAC8023**

  執行 8023 網路協定模組機制，裡面含有許多的 local timer 去切換傳送和接收的狀態。

- **Phy**

  這個模組實做有線網路在底層的行為。封包由這個模組開始排定封包傳輸事件。封包內的目的點 ID 在這裡決定。

- **Link**

  這個模組計算了相鄰點之間的鏈結延遲，並且我們在這個模組內初始化最短路徑表。

4.4.2 Wireless Network Protocol Modules

無線網路在 NCTUns 網路模擬器裡由八個模組組成。其中 WTCPDUMP、FIFO 模組並沒有修改，因為 WTCPDUMP 主要是統計之用，而 FIFO 不會排定任何事件到排程器裡。下面對修改的模組以簡短的方式做介紹。

- GOD

  這個模組模擬點對點網路 (ad-hoc network)繞路演算法。它會排定更新繞路表的本地運算事件，所以必需在此模組內為排定的事件加入來源點與目的點的 ID。

- MNode

  模擬 infrastructure mode 與 AP 溝通的機制。模組內存在許多本地計時器，

定時發送控制訊息給 AP。

- MAC80211

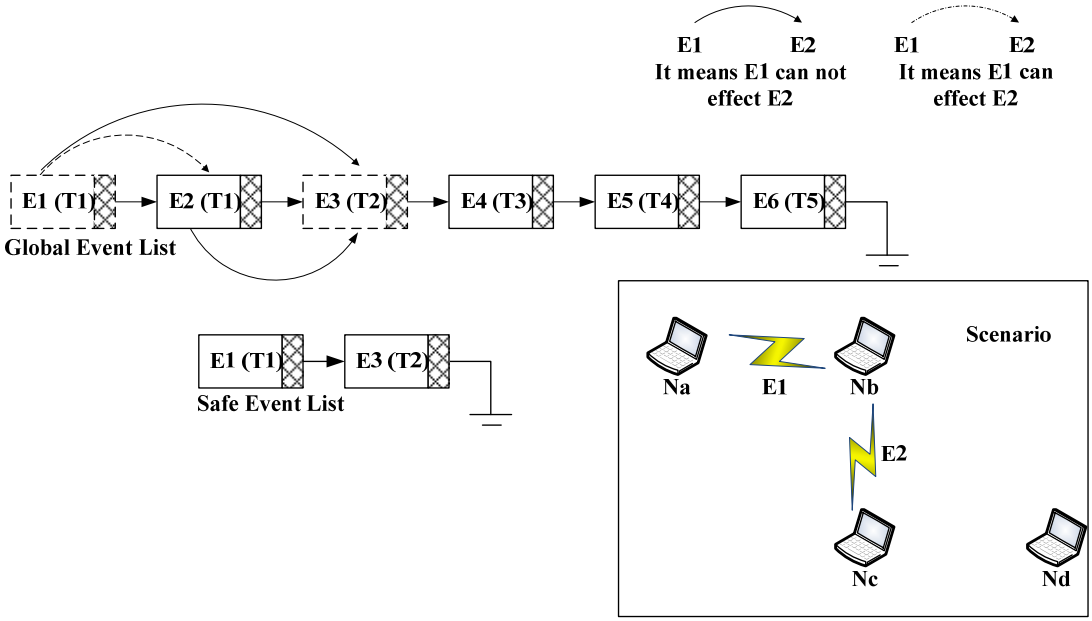
  實作 802.11b 網路協定的所有功能。模組內存在許多本地計時間，定時更改傳送及接收狀態。

- WPhy

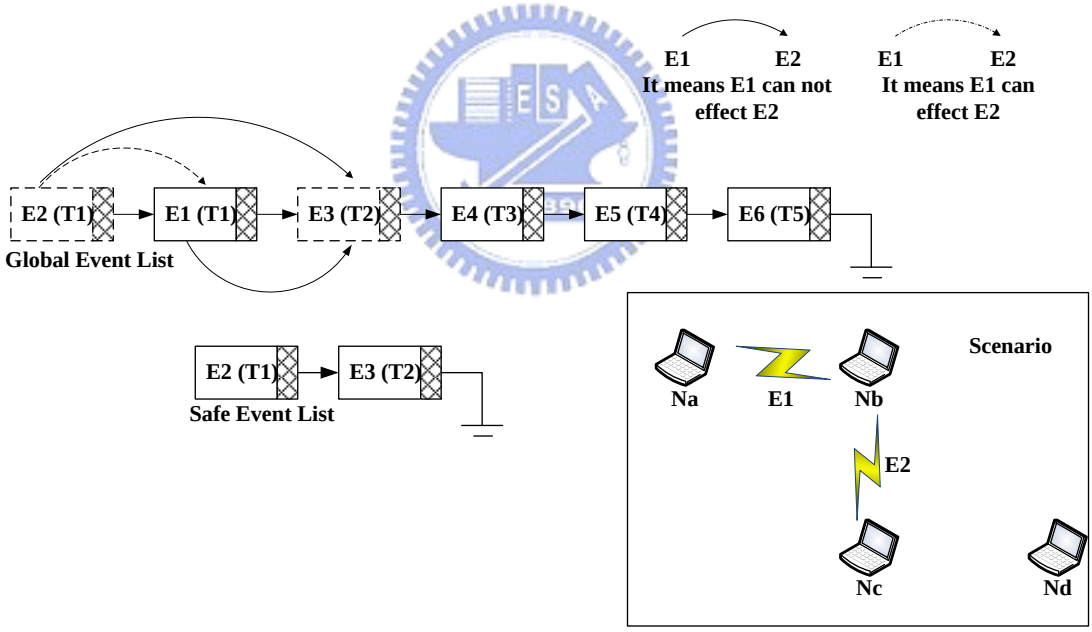
  模擬無線訊號傳遞的行為，包含能量的衰減、傳輸範圍計算，等…。並且由此模組開始排定封包傳輸事件。封包內的目的點 ID 在這裡決定。

4.4.3 A Wireless Network Boundary Problem

無線網路環境裡，各個節點是共用相同的無線通道傳輸封包。如 3.3 節敘述的問題，一個無線節點並無法通時傳送與接收封包，假設無線節點處在傳送狀態，如果同時間有別的節點傳送另一筆封包至目前的節點，兩筆封包將發生碰撞。另一方面，如果節點處在接收狀態，同一時間其它的節點再次傳送封包到目前的節點，會有兩種情況發生，假設目前正在接收的封包能量小於前一個已經收到的封包，正在接收的封包將被視為雜訊，並拋棄它。相反的，如果正在接收的封包能量大於前一個接收的封包，這種情況被視為碰撞 (collision)，兩個封包都必需被丟掉。兩筆封包前後執行的順序不同，將造成不同的結果，我們稱這樣的問題為 Wireless Network Boundary Problems。Wireless network boundary problems 無法使用 lock 機制去避免，因為這和事件在全域事件串列的順序有關。



(a) The E2 is inserted into global event list which is later than E1



(b) The E2 is inserted into global event list which is earlier than E1

Figure 4.13 Wireless Network Boudary Problem

我們將使用 Figure 4.13 解釋為什麼會產生 boundary 的問題。如 Figure 4.13 (a) 展示，在全域事件串列中，事件的安插順序為 E1、E2、E3…、E6。E1 與 E2 代

表的是相同時間 T1，封包分別從 Na 送往 Nb，和從 Nc 送給 Nb 的事件，由於 E1 在全域串列的位置在 E2 前面，所以 master thread 在找尋安全事件時，會將 E1 視為安全事件而先被執行。因為 E2 與 E1 的目的點相同，在條件檢測中會被視為不安全的事件，無法與 E1 同時執行。假設 E1 的能量高於 E2，則 E2 傳送的封包會被丟掉。相反的，如 Figure 4.13 (b)所示，假設全域事件串列內，事件安插順序變為 E2、E1、E3、…、E6。master thread 尋找安全事件過程，E2 會變成安全事件，E1 則為不安全的事件。由於 E1 的能量高於 E2，所以 E1 被執行時，E1 與 E2 傳輸的封包都會被丟棄。對於這個問題，目前沒有好的解法可以避免。因此，在下小節中，我們將提出方法驗證，當 Wireless Network Boundary Problems 發生時，模擬的結果並非是不正確的。

4.4.3.1 Validation

如同上小節中所提出來的問題，為了要確定 E1 與 E2 在循序模擬執行的順序，必需分析循序模擬過程中產生封包順序的排列，並將其套到事件層平行模擬方法，才能確保事件層平行模擬所得之結果與循序模擬相同。雖然這種作法可以保證，兩者的結果相同，但卻大大影響到事件平行度。因此，我們只是使用這個方式去驗證事件層平行方法只要遵照循序模擬封包執行順序，產生的結果就能與循序模擬相同。在第五章裡所要探討的模擬結果，和效能分析，並非使用這樣的方式執行得到的，而是透過多次的抽樣、平均得來的。

Chapter 5 ELP Performance Evaluation on NCTUns

事件層平行模擬方法為了可以在多核心系統下正常運作，它增加了許多額外的負載 (overhead)，包含全域變數必需上鎖，尋找可平行運算的安全事件，及 worker threads 與 master thread 的 IPC 通訊。上述因素都會關係到事件層平行方法可以提昇多少模擬執行的效能。由於複雜度的關係，我們僅量測尋找安全事件時必需花費的負載並使用 linux 系統提供的指令「top」去觀察每一條執行緒從模擬開始至結束總執行時間。

除此之外，單位時間內可以找到安全事件數目的多寡也同樣影響事件層平行模擬方法可否提昇執行效能的關鍵。因此，我們試著在各種網路環境下計算平均可搜尋到的安全事件數，此平均值在接下來的小節中都以 ELP Degree 表示。除了展示不同網路條件下最大執行的平行度外，我們也將比較循序模擬與事件層平行模擬在各種參數效能的比較。在觀察一些圖表之前，我們先對「效能提昇」、「負載」一詞提出較數學化的定義:

效能提昇 (Performance Speedup)

$$= \frac{\text{循序模擬執行時間}}{\text{事件層平行模擬執行時間}}$$

負載 (Overhead)

$$= \frac{\text{maste thread執行時間}}{\text{整體模擬時間}}$$

為了量測 CPU 使用率和使用不同機制 (循序模擬執、事件層平行模擬)下模擬執行時間，我們使用由 GNU project [9]提供的「time」指令。量測時所使用的硬體環境為 Intel(R) Core(TM)2 Quad 2.4GHZ CPU、主記憶體 1GHz，及 Linux

2.6.24.2 核心系統的四核心電腦。我們使用如上述規格的三台電腦進行測試，為了達到公平性，三台電腦使用相同的作業系統核心，無論是循序模擬或是事件層平行模擬都是建構在支援 SMP 架構的 Linux 核心系統，對於原本執行在單核心系統下的循序模擬在執行時間上可能或縮短一些，主要是因為其它的 CPU 可以代為處理額外非模擬引擎應用程式的要求，增加模擬引擎所獲得的 CPU 時間。

在模擬的過程中，可能會有許多非模擬引擎的應用程式請求 CPU 資源，為了避免這樣的情況發生，我們在模擬開始前會將絕大部份的應用程式關閉，但可能仍有部份的程式無法關閉，因此，執行的時間上可能會有些稍的誤差。

在接下來的小節中，我們將展示在有線網路和無線網路在不同條件下，ELP degree、效能提昇，及負載的變化趨勢。

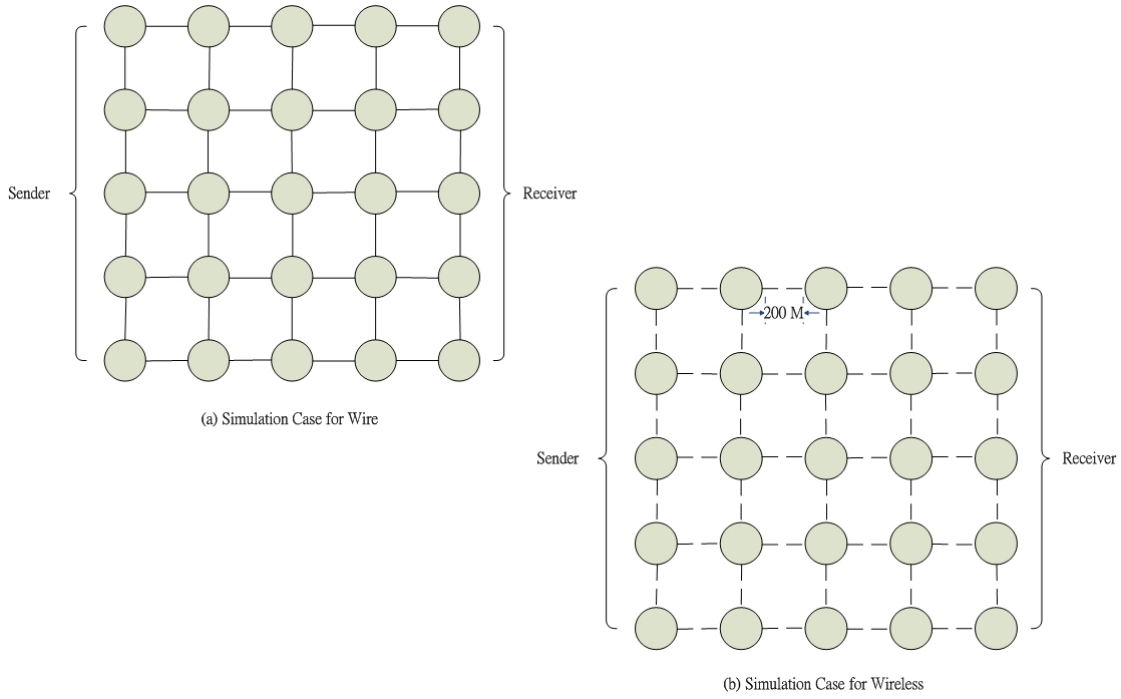


Figure 5.1 Simulation Topology Type

5.1 Performance Evaluation of Wired Networks

5.1.1 System Parameters

在 Table 5.1 中，我們列出所有使用在效能量測中的系統參數，不同的參數代表不一樣的網路環境。表中用粗、斜體表示的文字代表模擬過程中使用的預設值。下面將對這些系統參數提出更詳細的介紹及說明。

System Parameters	Value
Network Topology Size	5、6、7、8、9、***10***
Link Bandwdith	10Mb、***100Mb***、1000Mb
Link Delay	5ms、***10ms***、20ms、30ms
Coding Computation Loop	0、128K、256K、***512K***、1024K、2048k

Table 5.1 Wired Network System Parameters

用來量測效能的有線網路拓樸是由 NxN 格狀的點所組成，在此 N 是一個系統參數，代表「網路拓撲的大小」，它是介於 5 至 10 之間的整數。我們預設使用的拓樸大小是 10x10。每一個在格狀網路內的點與相鄰點之間都存在一條鏈結將其相連，如 Figure 5.1 所示。

鏈結延遲 (link delay)和鏈結頻寬 (link bandwidth)的值對於存在同一個網路模擬環境底下的所有鏈結都是相同的，並不會有 Na 到 Nb 的鏈結延遲是 10ms 而 Nb 到 Nc 的鏈結延遲為 5ms 的情況發生。同時它們也是效能量測探討中使用的系統參數。

鏈結延遲可以是 5ms、10ms、20ms 至 30ms，預設是使用 10ms。鏈結頻寬則從 10Mb、100Mb 至 1000Mb 中調整變化，其中使用 100Mb 做為預設值。傳輸

封包的大小，無論應用程式使用的是 TCP 協定或是 UDP 協定，都統一為 1400Byte。因為封包傳輸時間是封包長度除以鏈結頻寬，而封包長度是一個定值不會更改，所以，封包在鏈結上的傳輸時間是基於鏈結頻寬頻在變化。

使用在有線網路下的最後一個系統參數是一個名為「Coding Computation Loop」的變數，它代表的是程式在迴圈中執行的次數。我們藉由使用這個值呈現一筆封包編碼及解碼所花費的時間。它的數值可從 0、128K、256K、512K、1024K 至 2048K 變化。必需經過編碼及解碼的封包一定是準備傳往或接收 PHY 協定模組的封包傳輸事件，因此，經過「Coding Computation Loop」的事件勢必為封包傳輸事件。

所有測試的案例，不管是使用 TCP 或是 UDP 協定模組的應用程式，都會依據拓樸的大小決定使用應用程式的多寡。以 5x5 格狀網路來說，我們會使用十支應用程式，這些程式其中五支用來產生封包 (sender)被置於左側的點，另外五支則是用來接收封包 (receiver)對應到最右側的點。以此類推，當拓樸的大小為 10x10 時，同一時間將有 20 支程式同時執行。

除了使用不同的系統參數外，我們也會根據不同的協定調整模擬時間的大小。如果目前執行的案例使用的是 TCP 協定模組，模擬的時間設定為 100s，相反的，如果目前執行的案例是以 UDP 協定模組為主，模擬的時間將調整為 30s。為什麼 TCP 與 UDP 會使用不同的模擬時間呢? 因為 UDP 協定在是採用猛灌的方式在傳輸，所以，單位時間內由 UDP 產生的封包數量一定較 TCP 協定龐大。因此，在相同的模擬時間下必需花費更多的執行時間才能完成模擬，為了減少執行時間，我們刻意在使用 UDP 協定的網路環境底下，縮短模擬時間。在大部的模擬環境下預設的模擬時間為 100s。

無論使用循序模擬或是事件層平行模擬，我們所產生的每一個測試檔案所需的時間從 30 分到 1、2 天都有，由於一個模擬檔案執行的時間與事件的數量成正相關，為了確保事件層平行機制的效能的確較循序機制來的好，使用不同機制下

所產生的事件數應該必需是相同的，因此，我們試著去比較兩種機制下產生的紀錄檔，發現兩者產生的結果是相同的。

5.1.2 ELP Degree and Performance Speedups

為了量測模擬過程中同時執行的 CPU 數目，我們統計每一次 master thread 所尋找到的安全事件數，包含原本儲存在安全事件串列內的事件，並紀錄 master thread 從模擬開始到結束，安全事件的總搜尋次數，在模擬結束前，我們將此二值相除，得到一個平均值，代表此次模擬的最大平行度 (ELP Degree)，這也能讓我們更清楚看到，哪一類的參數對 master thread 搜尋安全事件有較大的關聯。

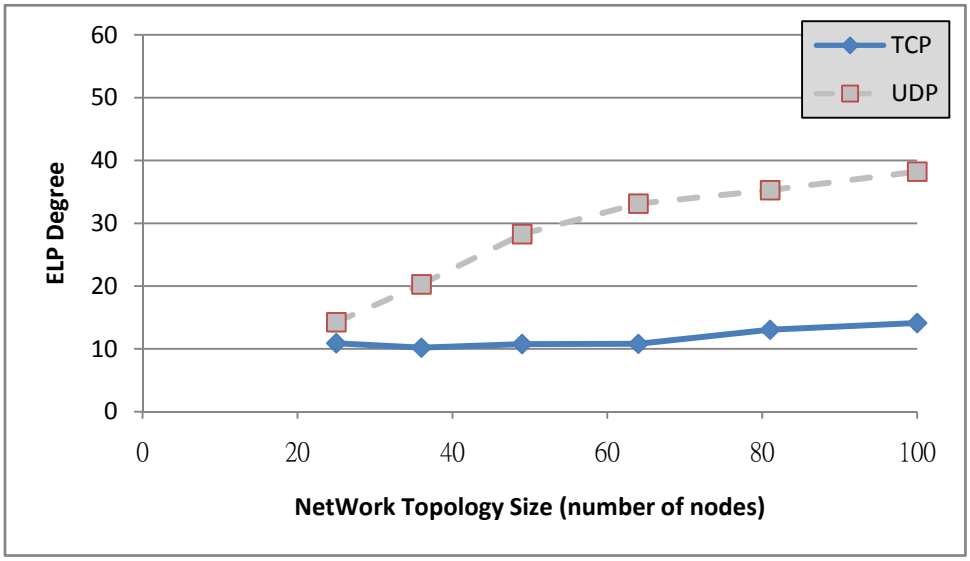


Figure 5.2 Wire: ELP Degree in the varied Network Topology Size

觀察 Figure 5.2，我們可以發現，隨著網路拓樸的增大，安全事件數目也相對增加。這樣的現象是能夠被解釋的，從 3.2 節討論的結果得知，只要符合 DstNID1 ≠ DstNID2，被檢測的事件就有很大的機率是安全事件，雖然符合 DstNID1 ≠ DstNID2的事件仍必需滿足 path lookahead 的計算，不過絕大部份只要能符合此條件的事件都可被平行運算。因此，隨著網路節點個數的增加，滿足此一條件的事件數也會隨之增加，所以，能找到的安全事件數也會跟著增加。這

對於大型網路是一個相當好的假設，因為一個龐大的網路往往模擬的時間必需要很長，如果隨著網路拓樸的增大，相對的也能找到更多的安全事件執行，那麼就可以讓模擬大型網路所需的時間也大大縮短。

從圖中我們可以觀察到 UDP 整體的 ELP degree 要比 TCP 大的多，這是因為 TCP 的 window size 有一個上限，而 UDP 是採取猛灌的方式在傳輸，所以，當頻寬為 100Mb/sec 時，TCP 最大流量大約是 42 Mb/sec，UDP 則為 91 Mb/sec，由流量的大小，我們已經可以很清楚看到使用 UDP 協定傳輸所得的事件數一定遠大於使用 TCP 協定。

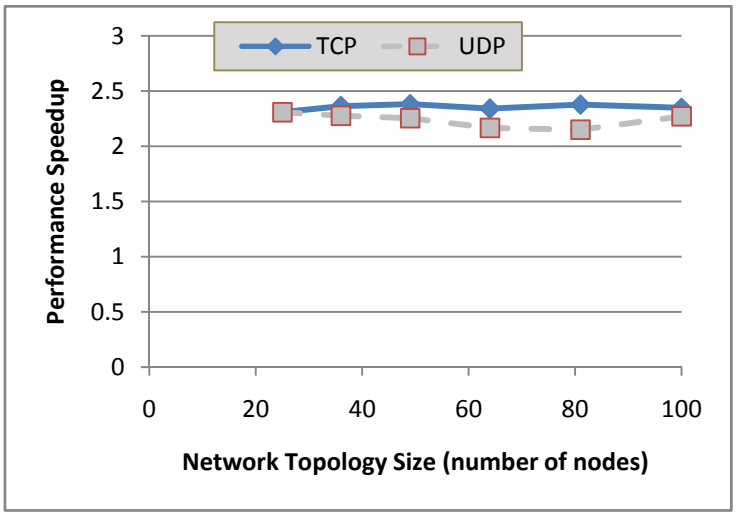


Figure 5.3 Wire: Performance Speedup in the varied Network Topology Size

從 Figure 5.3，我們可以觀察不同網路拓樸大小，使用 UDP 與 TCP 效能上的趨勢變化。對於 TCP，效能的變化都很穩定，幾乎都固定在接近 2.5 的數字上，而 UDP 則隨著網路節點個數的增加效能卻愈趨下降，主要是因為 UDP 的安全事件大部份運算的時間都很短，且網路拓樸加大，全域事件串列中的事件數也跟著增加，使得 master thread 尋找安全事件花費的時間更長，造成 worker thread 必需等待，浪費許多 CPU cycles。

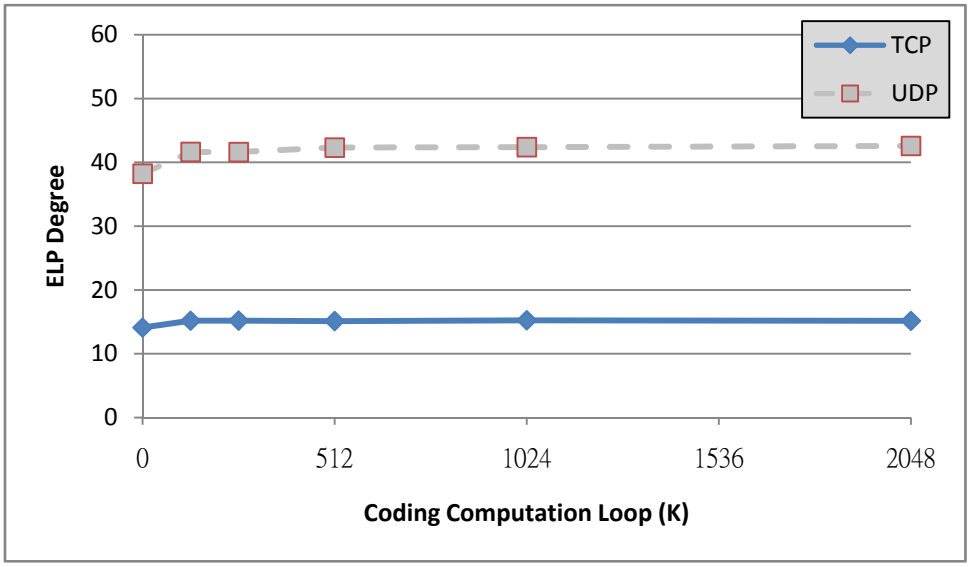


Figure 5.4 Wire: ELP Degree in the varied Coding Computation Loop

試著觀察 Figure 5.4，隨著「Coding Computation Loop」增加，ELP Degree 仍是很穩定的維持在一個定值，並不會隨著 coding loop 而產生變動，因此，從 Figure 5.5 中，我們很容易的可以得到當每一筆事件運算的時間愈長，速度提升的趨勢就愈加顯著，最主要是因為 ELP Degree 並沒有太大的變動，而事件運算的時間卻拉長，使得事件層平行模擬方法完成一筆事件所需的平均時間會比循序模擬短。這是一個很重要的概念，因為，如果事件運算的時間不夠長，導致 worker thread 很快完成安全事件的執行，造成 master thread 來不及搜尋足夠的安全事件供 worker thread 執行，就無法拿到較好的執行效能。

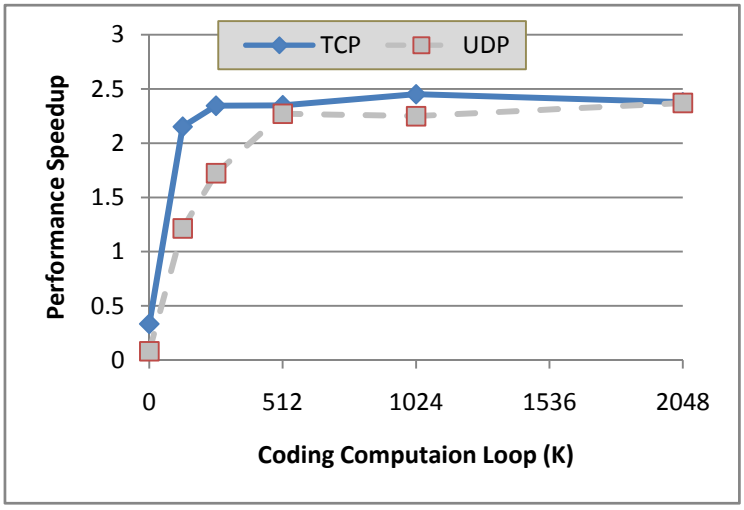


Figure 5.5 Wire: Performance Speedup in the varied Coding Computation Loop

由 Figure 5.6 可以看到，當頻寬愈大時，ELP Degree 就愈大，尤其是 UDP 協定。TCP 協定流量的多寡取決於 Sliding Window 的大小，因此 ELP Degree 的變化並沒有 UDP 顯著，而 UDP 每 50 us 會送 50 顆筆封包，只要頻寬足夠，它就能一直產生封包傳送給接收端，也因為這個原因，在相同時間內使用 UDP 協定得到的事件數遠遠高於 TCP。

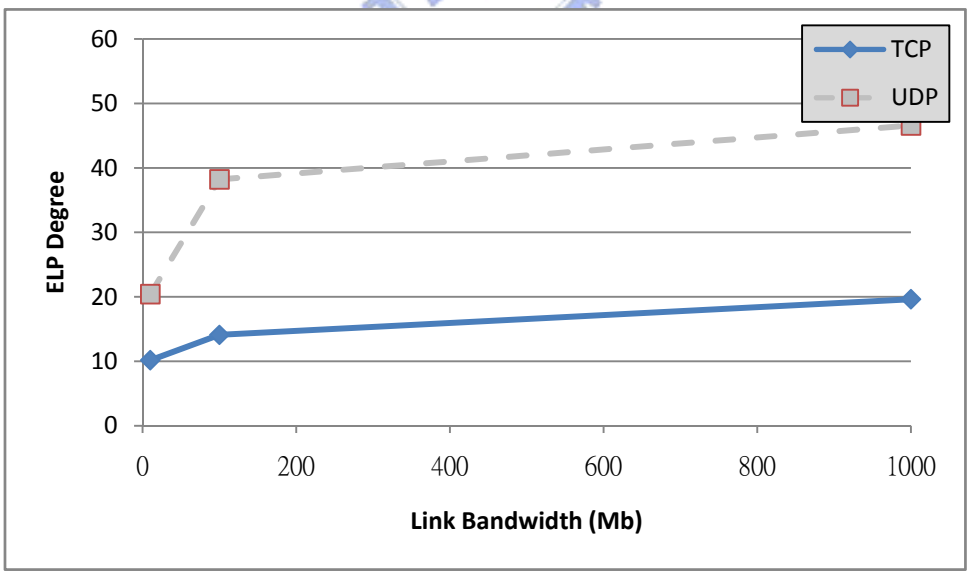


Figure 5.6 Wire: ELP Degree in the varied Bandwidth

由 Figure 5.7 看到隨著頻寬加大，不論是 TCP 或是 UDP 效能都不增反減，這在和探討網路拓樸大小變化時的原因相同，頻寬愈大，模擬過程中產生的事件數也愈多，但同時也使得 master thread 在找尋安全事件所花費的時間跟著拉長，這種情況在 UDP 更明顯，因此，我們可以觀察到 UDP 從頻寬 100Mb 到 1000Mb 的曲線中，有著比較顯著的下降。

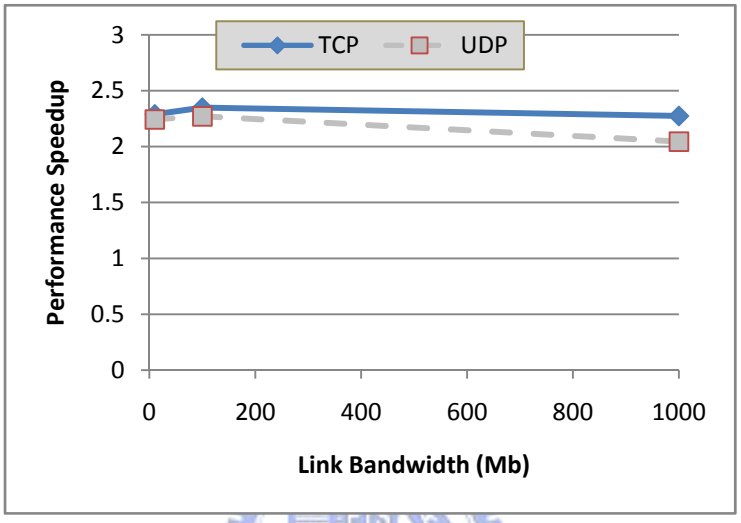


Figure 5.7 Wire: Performance Speedup in the varied Bandwidth

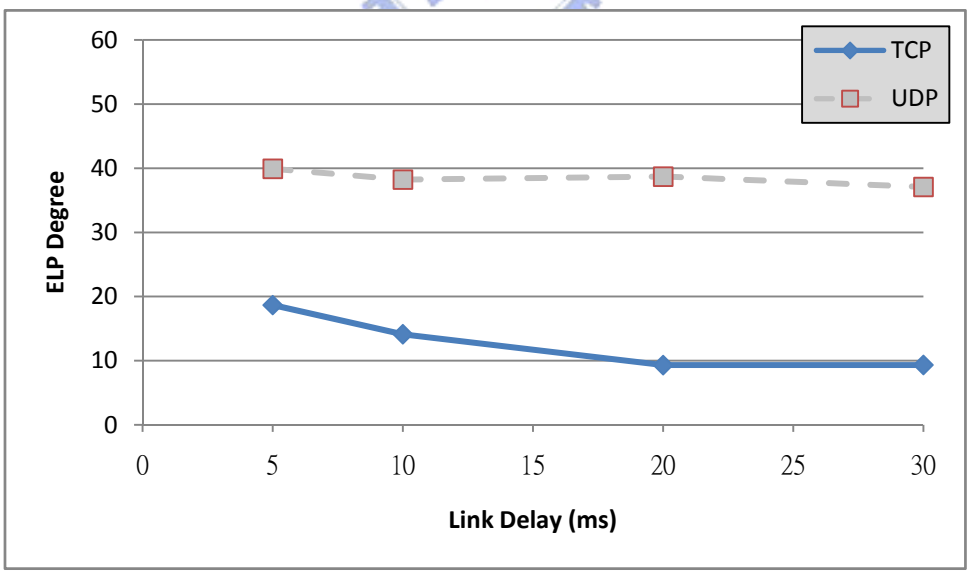


Figure 5.8 Wire: ELP Degree in the varied Link Delay

從 Figure 5.8 看到，模擬檔案中使用 TCP 協定，隨著鏈結延遲增加，可以找到的安全事件數目就愈少，這是因為 TCP 協定單位時間可以傳輸的封包量與 windows size 有關，而 windows size 要等到接收端的 ack 被傳輸端接到後才會增加，由於延遲加大，造成 ack 被收到的時間也延後，間接造成相同的模擬時間事件數銳減。

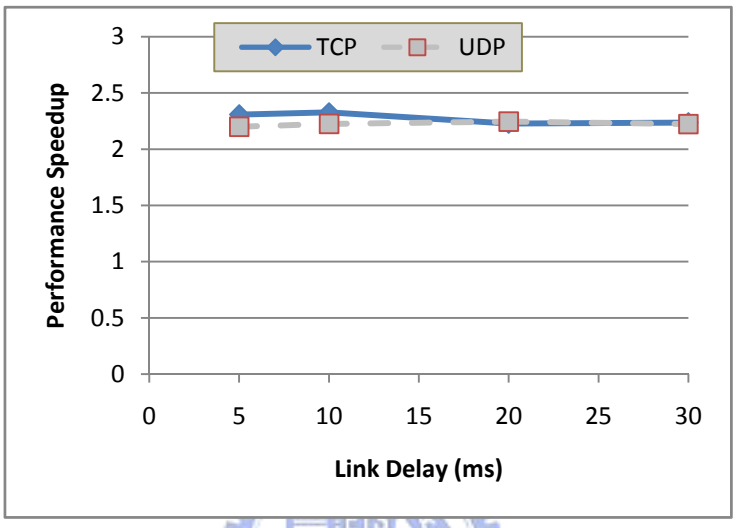


Figure 5.9 Wire: Performance Speedup in the varied Link Delay

從 Figure 5.9 可以看出，鏈結延遲對於效能影響的變化非常微小，但在傳統的平行模擬方法，加大延遲，等同於提高 path lookahead，會使的平行模擬的效能得到提昇，然而在我們的模擬結果中發現，延遲的大小並不會左右到模擬效能，這對於平行模擬來說是一個良好的條件，因為在鏈結延遲較小的無線網路環境中，效能上也能夠獲得提昇。

為什麼會有這樣的效果呢? 最主要的原因在於提高延遲相對的事件數也會跟著下降，因此，path lookahead 雖然增加，但可以搜尋的事件卻也縮減，如此一來，延遲增加的效果，被事件數的密度抵消，如同沒有增加延遲前的成果。

5.2 Performance Evaluation of Wireless Networks

5.2.1 System Parameters

在 Table 5.2 中，我們列出所有用在無線網路效能量測中的系統參數，不同的參數值代表相異的網路環境。表中用粗、斜體表示的文字代表模擬過程中使用的預設值。

System Parameters	Value
Network Topology Size	5、6、7、8、9、***10***
Link Bandwidth	1Mb、2Mb、5.5MB、***11MB***
Coding Computation Loop	0、128K、256K、***512K***、1024K、2048K

Table 5.2 Wireless Network System Parameters

與有線網路參數的種類相同，差別在於少了鏈結延遲 (Link Delay)的支援，因為在無線網路的環境中，兩節點的傳輸時間是由距離除以光速決定，所以，調整鏈結延遲等同於調整兩節點相對位置，在論文中，我們將相鄰兩點的距離設置為 200 公尺，不論是在何種網路條件，這個值都是固定不變的。

在無線網路的環境底下，我們使用如 Figure 5.1 (b)的拓樸做為模擬無線網路的環境，在拓樸上左右兩排的節點分別執行封包傳送、接收的應用程式，這些程式無論使用 UDP 協定或是 TCP 協定，模擬時間都固定為 100 秒，只有執行關於「Coding Computation Loop」模擬參數時，模擬時間才調整至 20 秒。

5.2.2 ELP Degree and Performance Speedups

與有線網路相同，我們仍是從 TCP 和 UDP 的角度觀察不同網路環境下 ELP Degree 的變化。

由 Figure 5.10，及 Figure 5.11 可以看出無線網路與有線網路在拓樸可變的條件底下，有著相似的結果，因此，我們能用有線網路在拓樸可變情況下的說明，解釋無線網路在相同條件下效能趨勢的走向。

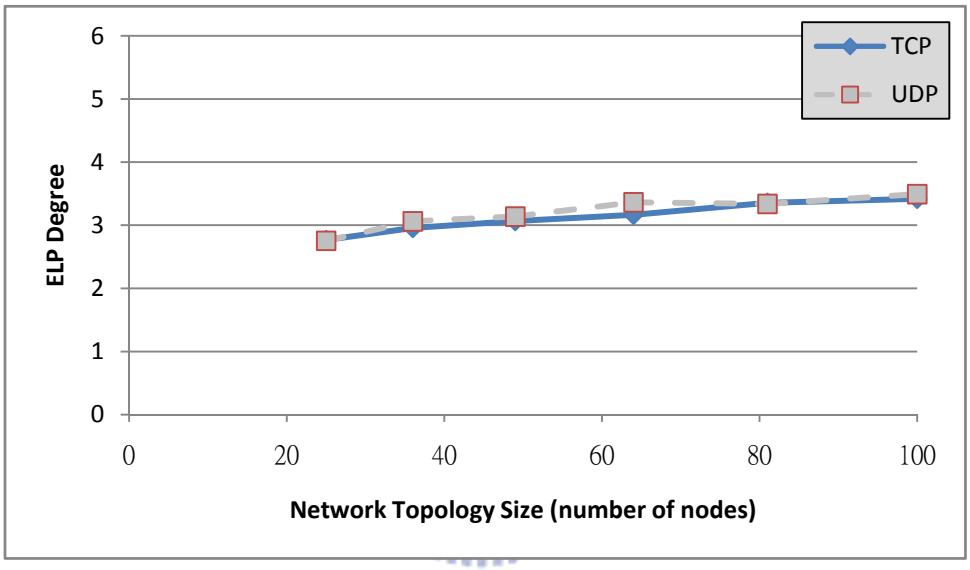


Figure 5.10 Wireless: ELP Degree in the varied Network Topology Size

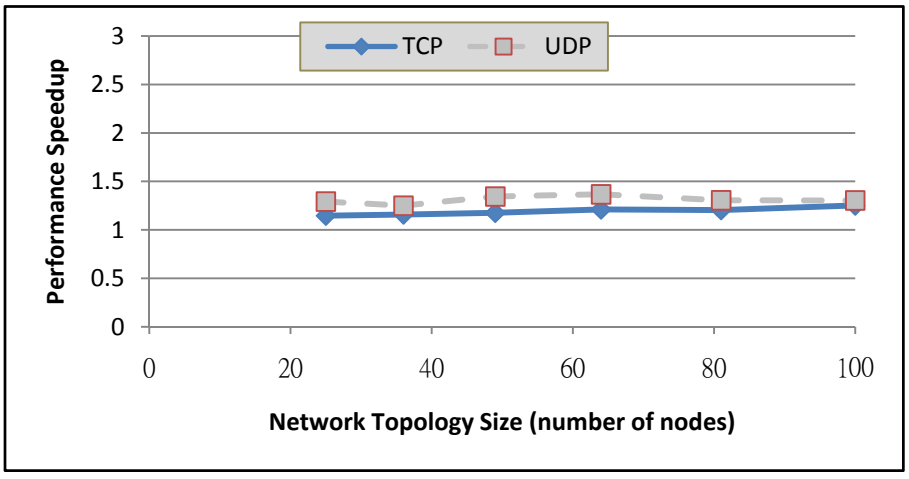


Figure 5.11 Wireless: Performance Speedup in the varied Network Topology Size

由 Figure 5.12 和 Figure 5.13 可以發現，隨著「Coding Computation Loop」的

增加，ELP Degree 沒有顯著的變化，但在效能增進上還是具有一定的成效，但與有線網路不同的是在無線網路環境下，「Coding Computation Loop」帶來的效益很快就達到一個上限，主要是因為無線網路環境底下，並不像有線網路有著較大的 path lookahead，造成 master 為了尋找相同個數的安全事件數，浪費的時間相對提高。

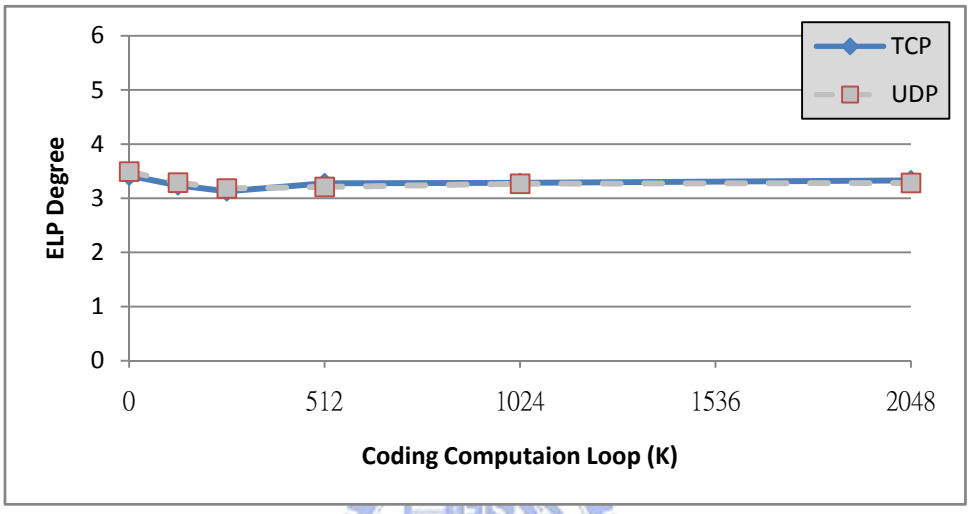


Figure 5.12 Wireless: ELP Degree in the varied Coding Computation Loop

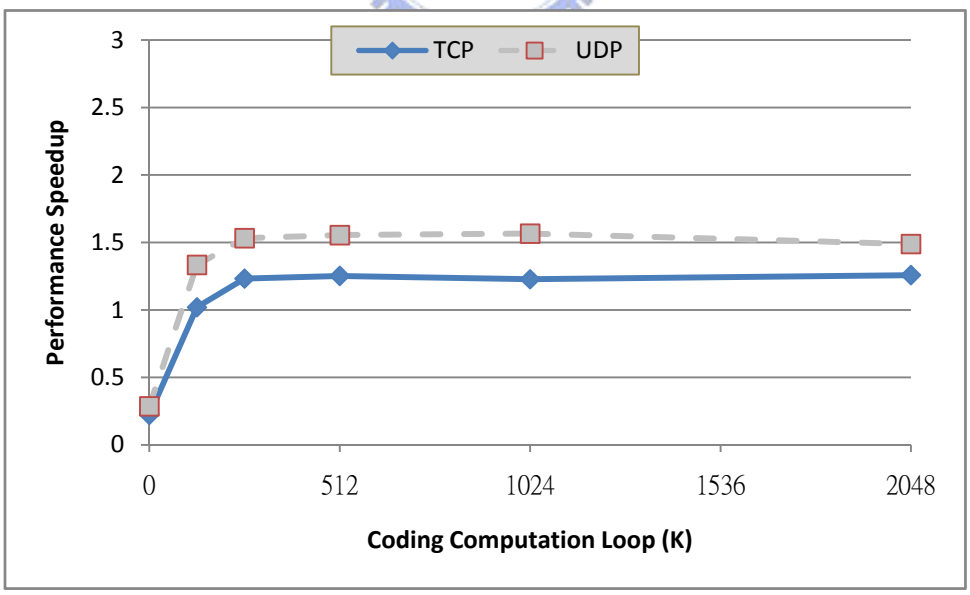


Figure 5.13 Wireless: Performance Speedup in the varied Coding Computation Loop

從 Figure 5.14，和 Figure 5.15 中能夠看到頻寬對於 ELP Degree 及執行效能都沒有顯著的影響，這是因為封包在無線通道傳送的時間太短，即使隨著頻寬的加大，使得事件個數增加，可以找到的同時執行的事件數仍是有限，因此，ELP Degree 並無大幅度的提昇。效能上也受 ELP Degree 的影響，並不隨著頻寬條件的改變而有太巨大的變化。

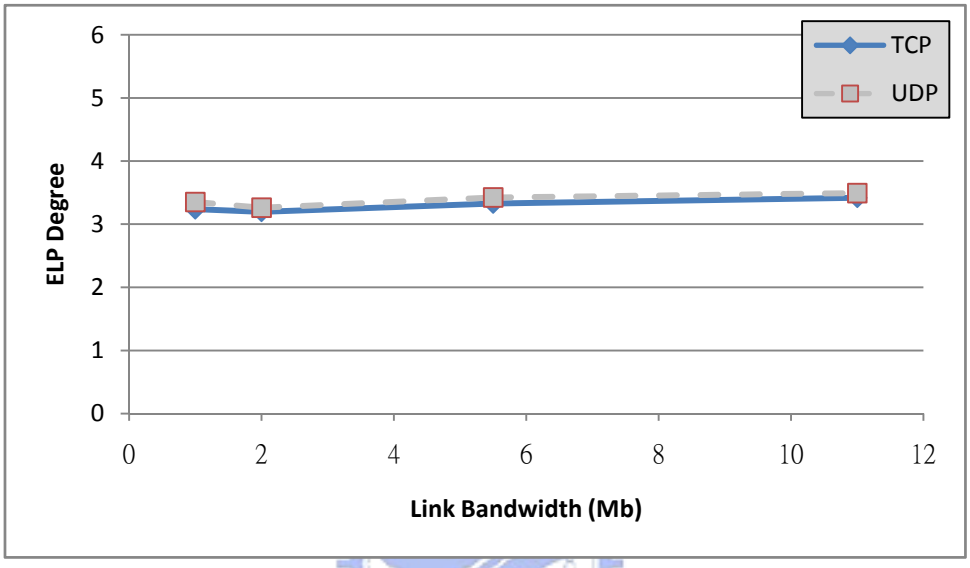


Figure 5.14 Wireless: ELP Degree in the varied Bandwidth

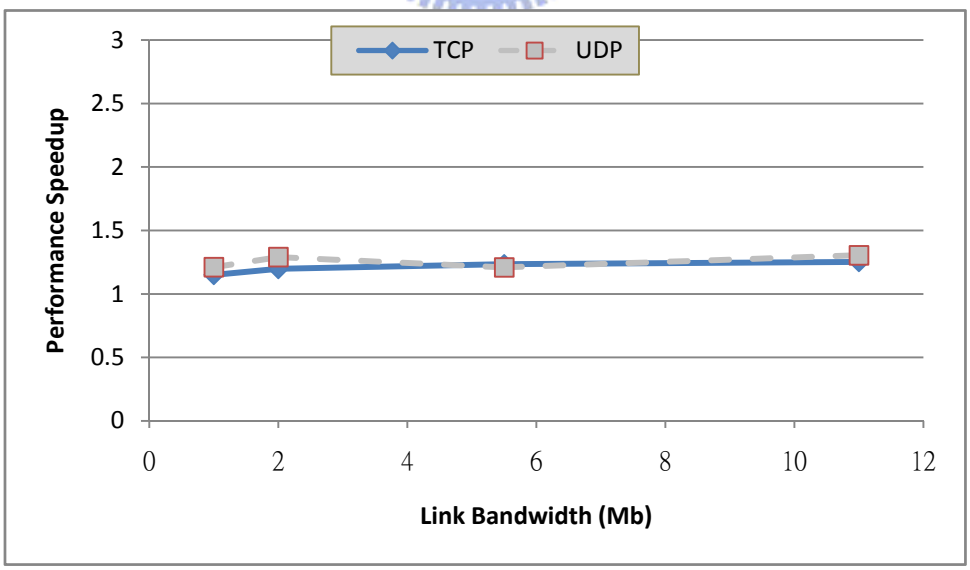


Figure 5.15 Wireless: Performance in the varied Bandwidth

Chapter 6 Discussion

在本章節中，我們將探討 NCTUns 網路模擬器使用事件層平行模擬方法的成果及限制，並對於部份由第五章效能分析所得的結果中，找出幾個影響效能成長的因素加以說明。

6.1 ELP Limitation on NCTUns

- Event Number

  從第五章的效能分析上，我們可以由「Network Topology Size」、和「Link Bandwidth」，這兩筆條件所做的量測發現，拓樸或是頻寬不大時，由於事件數量不足，master thread 無法在短時間內找到足夠多的安全事件，使得效能無法有顯著的提昇，不過隨著拓撲或是頻寬的增大，雖然事件的數量會增加，但是效能卻有下降的趨勢，這是因為存在全域事件串列的深度太長，造成尋找安全事件過程所需的時間變大。因此，我們能夠推得，不論事件數太多或太少對於事件層平行方法提昇效能都會有一定程度的影響。

- Coding Computation Loop

  「Coding Computation Loop」愈大，直覺上效能提昇的比例會愈加顯著，但由 Figure 5.5 展示，「Coding Computation Loop」的數目到達一定的程度，效能曲線就趨近於飽和，最主要是「Coding Computation Loop」加大，使得 worker thread 完成一筆安全事件的時間拉長，造成 master thread 在尋找安全事件的過程中，worker thread 可能仍在執行，如果正在執行的事件與事件串列中大多數的事件有著非常緊密的相依性，會使得 master thread 無法找到安全事件。

- Finding Safe Event Algorithm

  目前事件層平行方法，總是從全域事件串列的頭端開始檢測每筆事件的相依

關係來確保執行的事件不會影響到其它未執行的事件，造成「因果錯誤」發生，不過，這也造成全域事件串列深度愈深，尋找的事件必需花費的時間就愈長，所以，有效的改善搜尋的方法是必要的。

- NCTUns Architecture

NCTUns 網路模擬器無論是模擬引擎，或是協定模組，都存在許多相依的關係。如 4.4.3 節曾經介紹的 Wireless Boundary Problem，和在模擬引擎中，堆疊事件串列與時間串列如果具有相同時間需要執行的事件，堆疊事件必需先被完成才能執行時間事件，為了解決堆疊事件與時間事件執行的先後關係，在全域事件串列中，排列事件除了依靠時間戳記外，檢測事件屬應該被安插至哪一類的串列，也變成將事件插入到全域事件必需考慮的條件。除了上述的兩個問題外，NCTUns 仍有可能在修改其它協定模組遇到相似的問題，這也成為事件層平行方法整合至 NCTUns 網路模擬器中最怕遇到的難題。

Chapter 7 Future Work

- 支援各類型的協定模組

  在我們的設計上，目前僅支援 8023 有線網路及 80211-dcf 無線網路。由於事件層平行方法容易整合，對於未來支援各類型的協定模組是可以期待的，不過，對於移動式網路，使用事件層平行方法所需要考慮的層面較不會移動的網路要來的更多，最重要的一點是 master thread 必需不斷根據節點的移動狀況，計算各個節點的最短路徑，這也讓移動式的網路如果使用事件層平行方法需更多的負載。

- 增加自動切換循序模擬執行模式

  上一章節提到當事件平均運算時間小於使用事件層平行模擬必需花費額外的負載時，不會有顯著的效能提昇。因此，當事件平均運算時間小於額外的負載，事件層平行方法必需有能力切換至循序模擬，將 worker thread 的個數下降至一，並終止 master thread 尋找安全事件的行為。而當事件平均運算時間大於額外的負載，再自動切換回事件層平行模擬。

- 模擬引擎與應用程式同步執行

  第四章曾提到，整合事件層平行方法至 NCTUns 模擬器會出現模擬時間不同步的問題，同一章節中也提出問題的解法，不過使用此一解法並無法讓模擬引擎與應用程式同步執行，造成 CPU cycle 的浪費。為了令 CPU 使用率能更加提高，提出不同行程的同步執行是必要的。

- 支援 GUI 的操作

  原有的 NCTUns 模擬器支援文字介面與 GUI 介面的操作，但在我們的設計裡目前並不支援 GUI 的操作。NCTUns 模擬器運作在 GUI 介面時，會多執行三支不同的行程分別是 dispatcher、coordinator 和 GUI，worker threads 要如何與這些不同行程溝通將會是在 GUI 介面下使用事件層平行方法一定會

碰到的問題。

- 事件層平行方法應用在 N-core 系統的效能量測

  事件層平行方法可以很容易的隨著核心數的增加，調整執行緒的數目，達到最大的運算平行度，因此，在 N-core 系統中量測事件層平行模擬效能是值得期待的。不過，有可能一些硬體環境的限制，或是模擬網路拓樸的排列及作業系統的支援，我們認為事件層平行方法無法使得效能隨著核心數有著線性的成長。

Chapter 8 Conclusion

我們在論文中提出一套新穎的分散式平行模擬方法，並將它運用在 NCTUns 模擬器。它不同以往的方法，必需將原有的網路模擬器架構做大幅度的修改，而是從事件層面分析，每一筆事件是否相互獨立，進而決定它們彼此是否可以平行執行。因此，使用這套方法並不需要了解複雜的協定架構，只需在原有的事件結構內做小小的修改就能達到平行運算的效果，這也讓使用者不需為了將模擬器修改成可以平行運算，而額外學習平行模擬的概念。

在本文的各個章節中，我們介紹有關如何在 NCTUns 模擬器內使用事件層平行方法需做的修改，以及修改過程中遇到的問題和解決的方法。同時，我們也透過實驗結果的分析觀察到當每筆事件平均運算的時間大於使用事件層平行方法必需花費的額外負載時，執行的效能可以大幅提昇，相反的，如果每筆事件平均運算時間小於額外的負載時，效能提昇可能就沒有那麼顯著。

Bibliography

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

[2] Bader, D.A., Kanade, V., and Madduri, K. "*SWARM: A Parallel Programming Framework for Multicore Processors*" in the Proceedings of Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International. 26-20 March 2007.

[3] Richard M. Fujimoto. "*Parallel and Distributed Simulation Systems*". John Wiley & Sons, Inc, 200.

[4] Yue Li, and Depei Qian "*A Practical Approach for Constructing a Parallel Network Simulator*" in the Proceedings of Parallel and Distributed Computing, Applications and Technologies, 2003. PDCAT'2003. Proceedings of the Fourth International Conference. 27-29 Aug. 2003.

[5] NS2 http://www.isi.edu/nsnam/ns/.

[6] Thoppian, M., Venkatesan, S., Vu, H., Prakash, R., and Mittal, N. "*Improving Performance of Parallel Simulation Kernel for Wireless Network Simulations*" in Military Communications Conference, 2006. MILCOM 2006**.**

[7] Bagrodia R. X. Zeng and M. Gerla. "*GloMoSim: A library for the parallel simulation of large wireless networks*". Workshop on Parallel and Distributed Simulation 1998, 154-161.

[8] Wikipedia http://en.wikipedia.org/wiki/Native_POSIX_Thread_Library.

[9] GNU http://www.gnu.org/.

[10] Y.S. Tzeng. "*The Performance of the NS2 Network Simulator using the Event-level Parallelism Approach*". 2007.

[11] Daniel P. Bovet and Macro Cesati "*Understanding the Linux Kernel*" O'Reilly November 2005.

[12] The Linux Kernel www.kernel.org.

[13] Completely Fair Scheduler (CFS) http://www.ibm.com/developerworks/linux/library/l-cfs/index.html.

[14] S.Y. Wang, C.H. Huang, C.C. Lin, C.L. Chou, and K.C. Liao."*The Protocol Developer Manual for the NCTUns 4.0 Network Simulator and Emulator*".

[15] 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.