國立交通大學

資訊科學與工程研究所

碩士論文

在多核心系統下提升 NCTUns 的效能

Improving Throughput of NCTUns on Multi-core Systems

研 究 生：曾楷文

指導教授：王協源　教授

中 華 民 國 一 百 年 九 月

在多核心系統下提升 NCTUns 的效能

Improving Throughput of NCTUns on Multi-core Systems

研 究 生：曾楷文　　Student：Kai-Wei Tseng

指導教授：王協源　　Advisor：Shie-Yuan Wang

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

September 2011

Hsinchu, Taiwan, Republic of China

中華民國一百年九月

在多核心系統下提升 NCTUns 的效能

Improving Throughput of NCTUns on Multi-core Systems

研究生：曾楷文　　指導教授：王協源

國立交通大學

資訊科學與工程研究所

摘要

NCTUns 是一款高知名度且功能強大的網路模擬器。它所使用的特有模擬方法為它帶來了高可信度的結果，但也限制其無法在 SMP 架構的系統下執行。為此我們提出了新的設計，修改 NCTUns 能在 SMP 架構系統下執行並支援並行模擬。

在本篇論文中，我們將首先介紹 NCTUns 的設計架構，接著詳述我們提出的設計與實作修改，然後比較原始版本與支援並行模擬版本 NCTUns 的效能，最後討論與以虛擬機器方式實現的並行模擬之間的優缺。

關鍵字：網路模擬、NCTUns，並行模擬，虛擬機器

Improving Throughput of NCTUns on Multi-core Systems

Student: Kai-Wei Tseng Advisor: Shie-Yuan Wang

Institute of Computer Science and Engineering

National Chiao Tung University

ABSTRACT

NCTUns is a well-known and powerful network simulator. The unique simulation methodology applied on NCTUns brings high confidence results but also limits it cannot be executed on SMP systems. Therefore, we propose a new design of NCTUns supporting concurrent simulation on SMP systems.

In this paper, we first introduce the basic design of NCTUns. Then we describe our design and implementation. Next we compare the performance to the original NCTUns. Finally, we discuss the possibility and advantage of supporting concurrent simulation by virtual machine.

Keywords: network simulation, NCTUns,

concurrent simulation, virtual machine

誌謝

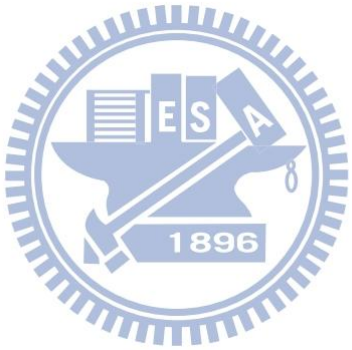
感謝我的指導教授王協源教授在這兩年來對我的悉心教導，令我在專業領域甚或工作態度上學習到許多。

感謝林華君教授、吳曉光教授以及黃仁竑教授對我的論文的建議與指導，讓這篇論文更加完善。

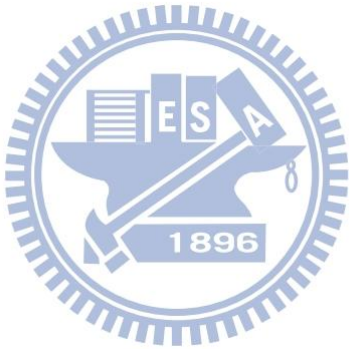
感謝所有的實驗室成員，這是很美好的學習環境。

感謝我的家人對我的支持。

我畢業了。



目錄



表目錄

圖目錄

壹、緒論

1.1 研究動機

網路模擬在進行網路相關研究時是經常被運用的技術，利用模擬的技術來驗證能較實際佈建網路拓樸節省成本，也比單純數學推導來的更具可信度。

而 NCTU network simulator(以下均簡稱 NCTUns)正是一款知名的網路模擬器。由王協源教授帶領交大的團隊開發維護，基於其獨特的 kernel-reentering 模擬方法，有著下列其他模擬器沒有的優點。是許多研究或學習網路者選擇的工具。

- 它直接使用現實生活中的 TCP/IP protocol stack 參與模擬，所以能產出高可信度的模擬結果。
- 它支援現實生活中的任意程式能夠直接在模擬世界裡執行，與模擬世界的 node 互動。
- 它提供高整合度的 GUI，使用者可以非常方便的利用於設定模擬任務，而不需要去手動修改檔案。

然而在 NCTUns 現有的設計並沒有支援 kernel 開啟 SMP，也就是說其只支援單核心的作業環境。如今 CPU 的發展趨勢已經不是一味地追求提高時脈，而是朝提高核心數的方向發展，目前市場上已經難見單核心的 CPU，這麼一來當我們使用一台多核心 CPU 的主機運行 NCTUns，實際上卻只會利用到單一核心，其餘核心等同於閒置，這是很嚴重的系統資源浪費。本篇論文的動機便是將這些浪費的資源再利用。

1.2 研究目標

為了利用在多核心系統環境下執行 NCTUns 時閒置的 CPU，我們目標是修改 NCTUns 使其能夠在支援 SMP 的 kernel 下正確模擬，如此除了負責 NCTUns 模擬工作的 CPU，其餘 CPU 也能分擔一些系統工作。但是讓剩餘的 CPU 僅負擔平常的系統工作其實也是一種浪費，因為系統工作並不需要太多的 CPU cycle，不如想個方式利用其餘的 CPU 來增進 NCTUns 模擬的效能。

我們觀察到最近雲端技術的興起，假設現在建立一座雲端模擬中心並購入許多高階的伺服器來向大眾提供模擬服務，如果還是使用原本的 NCTUns，一台再高階的伺服器也只能提供一個用戶模擬服務。

於是若我們開發出支援並行模擬的 NCTUns，就能夠充分利用高階伺服器的優勢，讓一台高階伺服器同時提供多人的模擬服務。這也就是本篇論文的目標，發展出一套能在 SMP kernel 下支援並行模擬的 NCTUns。

1.3 章節概述

在接下來的章節裡，第二章將介紹 NCTUns 的基本設計概念與運作流程以期讀者能了解原本 NCTUns 的優點及限制，並能夠在之後閱讀第三章節實作部份時，能清楚為何需要修改及為何如此實作；第三章將指出 NCTUns 在多核心環境下的問題，並提出要發展出一套可並行模擬的 NCTUns 所需要的設計修改與實作；第四章則是比較我們實作完成的成品與原始版本 NCTUns 的效能；第五章則針對當使用虛擬機器進行並行模擬與使用我們的設計的優缺比較；第六章為我們這篇論文簡單下個結論；第七章則是提述在撰寫論文過程中，我們所發現未來能夠加強之處。

貳、背景

2.1 NCTUns 主要元件

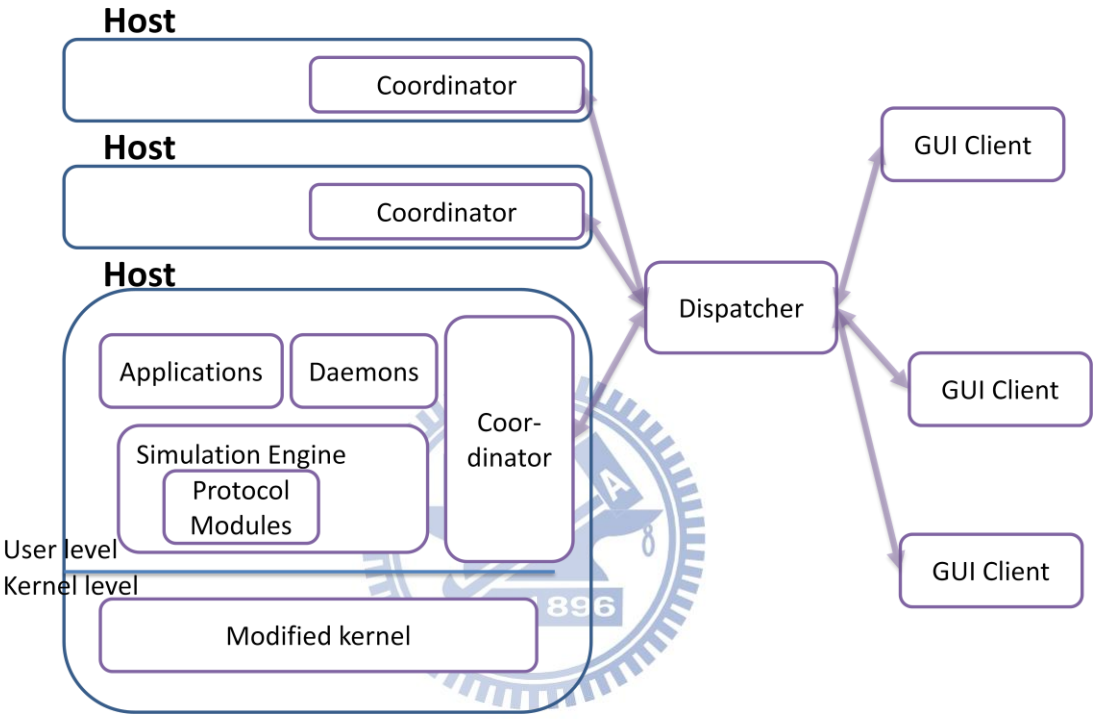


圖 2.1 NCTUns 的主要元件

NCTUns 由八個主要元件構成，以下一一詳述。

GUI Client：是一隻提供給使用者操作的圖形化介面程式，使用者能夠藉此佈建網路拓樸，調整模擬設定例如 mobile node 的移動路徑，link 的頻寬，決定在模擬世界裡執行哪些應用程式等等，並允許對 simulation engine 下命令，在模擬結束後也能觀看封包的動畫或是種種統計資料。可以說使用者能夠完全使用 GUI Client 這支程式直觀且方便地來完成整個模擬操作而不需要手動再去修改其他檔案。

Simulation engine：是一隻 user level 的程式，是整個模擬運行的核心，可視為一小型的作業系統，負責事件的排程，模擬時間的管理等多樣功能，以保證模擬順暢運行。結合 protocol modules 後負責整個模擬工作，以 GUI 所產生的設定檔作為輸入，運行模擬後再輸出結果檔案提供 GUI 展示。

Protocol modules：由 NCTUns 提供用以堆疊 protocol stack，一般來說一個 protocol module 代表一項 protocol 或一項功能，例如 Address Resolution Protocol 與 First-In-First-Out 都被分別寫為 ARP module 與 FIFO module，使用者能夠使用 NCTUns 已提供的 protocol module 或是自己撰寫，然後任意的組合 protocol module 堆疊出 protocol stack，封包將依序經過這些 protocol module 的處理達到模擬的目的。

Dispatcher：是一隻 user level 的程式，是 NCTUns 的分散式架構裡的核心，如圖所示一隻 dispatcher 能夠管理多個 coordinator 與多個 GUI client，是必須要最先執行的程式，且 dispatcher、coordinator、GUI client 三者可以不需要執行在同一台機器上，使用者可以只使用 GUI client 程式完成模擬工作的設定，再送往遠端進行模擬，最後由遠端傳回結果。當 GUI client 下了進行模擬命令，便會先詢問 dispatcher，dispatcher 則會分配空閒的 simulation engine 給要求的 GUI client 提供模擬服務，如果沒有空閒的 simulation engine，dispatcher 會將模擬要求先保存在自身的佇列中等待分配空閒的 simulation engine 處理。

Coordinator：也是一隻 user level 的程式，代表 simulation engine 與 dispatcher 與 GUI 互動，主要是標示 simulation engine 是正在進行模擬亦或

可提供模擬。而在接收到模擬請求時，fork 出 simulation engine 來提供服務。

Kernel Modifications：由於 NCTUns 獨特的模擬機制，能夠直接使用現實生活中的 TCP/IP protocol stack 參與模擬以提供更高的模擬可信度，因此需要修改 kernel 才能讓 kernel 能正確與 simulation engine 互動。

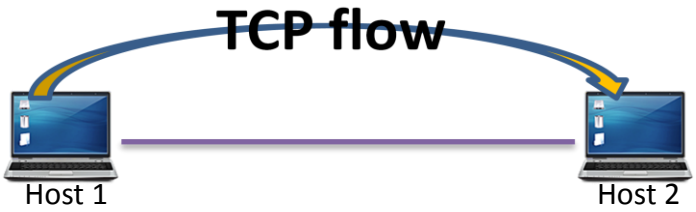
Daemons：一些提供特定功能幫助模擬的 user level 程式，例如 RIP routing daemons 或是 OSPF routing daemons 就是在模擬時運行在模擬的 router 上，分別以 RIP 與 OSPF 兩種 routing protocol 在 router 之間交換資訊並更新 routing table，達到模擬 router 的效果。

Applications：使用者可以藉由執行它們產生需要的流量，NCTUns 提供了許多現成的 Applications 例如 stg、rtg，它們可以根據需求創造 TCP 或是 UDP 不同 protocol 不同大小的背景流量。除了 NCTUns 提供的流量產生程式，使用者也能夠不加更改直接使用現實生活的任意程式作為產生流量的工具，這點在測試是非常方便的；例如在相接的兩個模擬 node 上面分別執行 vlc server 與 vlc client，變更模擬的參數例如封包遺失率再觀察影像傳輸的品質。

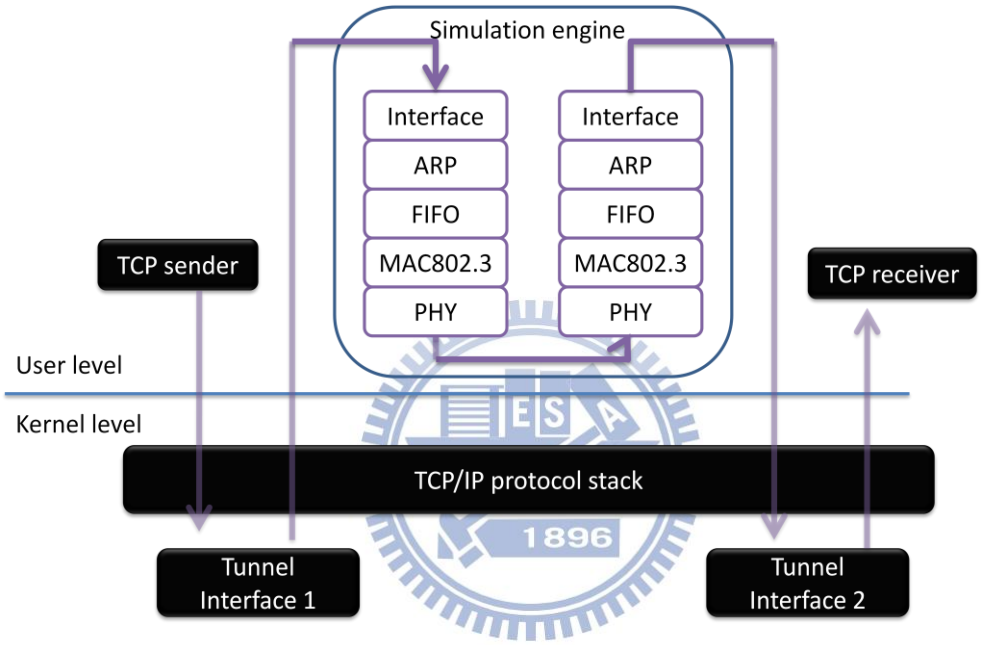
2.2 Kernel Re-entering 模擬機制

前面敘述到 NCTUns 的優點有因為能夠直接使用現實生活中的 TCP/IP protocol stack 所帶來的模擬結果可靠性，以及能夠直接在模擬世界中使用現實生活的應用程式來產生流量所帶來的方便性，這些都是因為 NCTUns

採用了一套特殊的模擬方法，我們稱之為 kernel re-entering 模擬機制。



(a) 模擬網路拓樸



(b) 模擬封包處理流程

圖 2.2 kernel re-entering 模擬機制

見圖 2.2a，這是在模擬世界裡的網路拓樸，使用者建立兩台以 Ethernet link 相接的 Host，而 Host1 上執行一隻 TCP sender 程式，Host2 上執行一隻 TCP receiver 程式，兩者之間建立 TCP 連線不斷傳送資料。我們以此為例解釋 kernel re-entering 模擬機制如何運作，模擬封包自 Host1 傳輸到 Host2。

見圖 2.2b，封包由 user level 的 TCP sender 產生，並如一般處理封包流程送入 kernel 由 kernel 裡的 TCP/IP protocol stack 處理，首先進入 TCP layer 的資料會先被包上 TCP header，再送入 IP layer 包上 IP header，進入 routing

的程序，由於我們特殊設計的 routing 機制，封包並不會被送到真實存在的網卡，而是被送入由 NCTUns 所建立的 tunnel interface 1，接下來 simulation engine 就會由代表 tunnel interface 1 的 interface module 中自 kernel level 取得封包內容。

注意這個時候封包並沒有由真實的網卡出去，歷經真實的傳輸狀況，而是被抓回了 user level 交由我們設定的 protocol stack 來模擬 datalink layer 與 physic layer 的行為。如圖 2.2b 所示，在 interface module 取得封包後，經過 ARP module 實現 ARP protocol 反查出目標 node 的 MAC 位址，再經過 FIFO module 模擬網卡的 output queue，然後是 MAC802.3 module 模擬 IEEE 802.3 MAC protocol(CSMA/CD)，最後 PHY module 則是負責將封包送至另一端的 PHY module 並實現傳輸延遲；例如將這條 link 的傳輸延遲設定為三秒，那麼另一端的 PHY module 就必須等待模擬時間推進三秒才能繼續模擬。收端同樣在 user level 利用 module stack 實現模擬，當最後封包往上丟回代表 tunnel interface 2 的 interface module 時，此 interface module 會將封包寫向 tunnel interface 2 並且通知 kernel。

Kernel 開始處理 tunnel interface 進來的封包，對 kernel 而言就像是處理一般網卡進來的封包，同樣丟回 IP layer 解開 IP header，再丟回 TCP layer 解開 TCP header，最後丟回 user level 由 TCP receiver 接收；完成整個傳輸過程。

為了利用現實存在的 TCP/IP protocol stack，在這整個模擬過程中封包一共進出 kernel 兩次，這正是取名為 kernel re-entering 的原因。而也正是因為利用現實存在的 TCP/IP protocol stack，而不是自行寫程式模擬 TCP/IP protocol 處理封包的行為，現實生活的應用程式不需要經過修改就能夠直接在 NCTUns 的模擬世界上執行。

2.3 S.S.D.D routing 機制

在前面解釋的 kernel re-entering 模擬機制裡，很重要的一點是能夠在封包進入 IP layer 時，能將封包送至相應的由 NCTUns 產生的 tunnel interface。為了保持 IP protocol 的真實性，我們其實並不希望對 IP protocol 作大改動，因此在這裡是以添加 routing entry 的方式來讓封包在 routing 時，被相應的 routing rule 丟到 NCTUns 的 tunnel interface。

Dst	Iface
1.0.1.2	tun 1
1.0.2.1	tun 1
1.0.2.2	tun 1

Dst	Iface
1.0.1.1	tun 3
1.0.2.1	tun 4

Dst	Iface
1.0.2.2	tun 2
1.0.1.1	tun 2
1.0.1.2	tun 2

圖 2.3 routing entry 衝突示意圖

見圖 2.3，我們以兩台 host 中接一個 router，並且互相以 Ethernet link 相連這樣的例子來說明 NCTUns 的 routing 機制。首先 GUI 讀入使用者的模擬設定之後，會自動分配每個 interface tunnel ID 以及 IP address，並根據分配下去的 tunnel id 與 IP address 自動生成所需的 routing rule。

在圖 2.3 每一個 node 下列出的便是它們將在 routing table 裡建立的 routing entry。舉左邊這個 host 為例，當它想送出一個目的地 IP 為 1.0.2.1 的封包，那麼 routing 的時候就會查詢到左下第二條的 routing entry 並得知應該把封包丟到 tun1 這個 tunnel interface。

這樣的設計其實有一個問題就是 routing entry 的衝突，注意僅管在模擬世界裡有許多 node，但在現實世界 kernel 卻只有一個，也就是只有一張 routing table，因此當全部的 routing entry 加下去，請見左邊 host 的第二條

routing entry 與 router 的第二條 routing entry，這兩條 entry 都在規範目的地 IP 是 1.0.2.1 的封包，然而指定丟出的 interface 卻分別是 tun1 與 tun3，這樣的 routing entry 衝突會使得 kernel 不知道如何 routing。

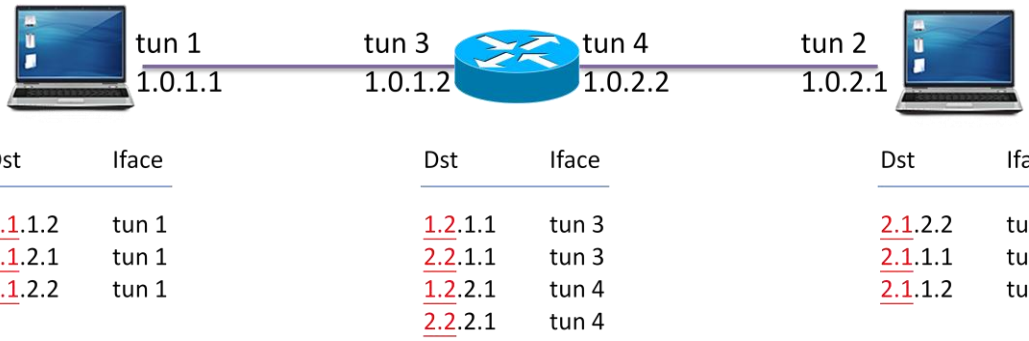


圖 2.4 routing entry 無衝突示意圖

為了解決這樣的問題，NCTUns 提出了一套 S.S.D.D routing 機制。觀察上述 routing entry 衝突的問題，會發生錯誤是因為對於同樣目的地的封包，不同的 node 處理時發現需要丟向不同的 tunnel interface。因此 NCTUns 在這裡將加入的 routing entry 的目的地 IP address 欄位作了 S.S.D.D 轉換。

所謂的 S.S.D.D 指的是將原本封包裡的 IP address 在丟進 kernel 處理時轉換為 source IP pair 和 destination IP pair 的組合，限制 GUI 在分配 IP address 時只能分配 1.0.a.b 形式的 IP，因為前兩位要在作 S.S.D.D 時用以填充 source IP pair，不能拿來分配；而分配的.a.b 才是 NCTUns 模擬世界裡真正能區別 IP address 的部分，所以上述的 source IP pair 與 destination IP pair 自然就是 source IP 的後兩 byte 與 destination IP 的後兩 byte，也就是.a.b 的部分。

原本由不同的模擬 node 新增的目的地 IP address 重複的 routing entry，由於屬於不同模擬 node，所以 source IP 必然也不同，經過 S.S.D.D 轉換後，就會出現獨一無二的 IP，就能夠避免 routing entry 的衝突。

同樣舉左邊這個 host 為例，當它想送出一個目的地 IP 為 1.0.2.1 的封包，

經過 S.S.D.D 轉換，kernel 看該封包的目的地 IP address 就會是 1.1.2.1，前面的 1.1 來自丟出封包 host 自身的 IP address 1.0.1.1，後面的.2.1 則是來自目的地 IP1.0.2.1，那麼 routing 的時候就會查詢到左下第二條的 routing entry 並得知應該把封包丟到 tun1 這個 tunnel interface。

2.4 Discrete event 模擬方法

NCTUns 整個的模擬過程可以說是由一連串的模擬 event 所組成，回顧 NCTUns 的主要元件，在模擬過程中 Applications 會產生 event，Daemons 會產生 event，Protocol modules 與 kernel 也會產生 event，這麼多的 event 該如何決定哪些 event 要先處理，哪些 event 可以之後處理？又是誰來負責做這些決定？

在 event 的資料結構裡，都會有一份觸發時間，這份觸發時間依循的標準是模擬時間而不是現實時間。全部 event 的觸發順序就應該依照觸發時間依序觸發。而模擬時間又是由誰來維護？

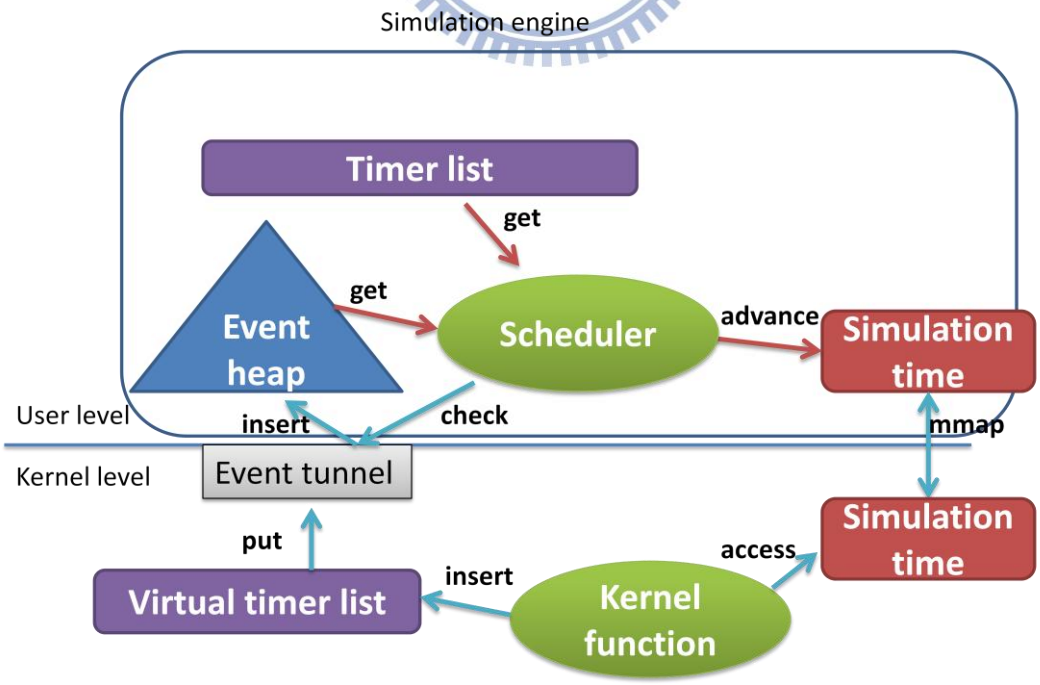


圖 2.5 discrete event simulation

Simulation engine 就扮演了這麼一個重要的角色，負責維護模擬時間並且根據模擬時間來依序觸發 event。我們用上圖 2.5 來說明 simulation engine 處理的流程。

首先解釋單純一點的情況，暫不考慮與 kernel 互動也就是不存在 kernel event，先看圖中 user level 的部分，event heap 與 timer list 內存由可能 Applications、Daemons 或 Protocol modules 發出的 event 或 timer，timer 亦可視為 event 的一種，前段敘述僅提 event 為概括而論。儲存在 event heap 與 timer list 裡的 event 或 timer 會依照觸發時間的先後排序，離現在的模擬時間最近的，會放在 list head 或 heap root。

simulation engine 裡的 scheduler 會不斷地去 event heap 與 timer list 中取出觸發時間離現在模擬時間最近的 event 或 timer，取出之後則先將模擬時間推進到該正要處理的 event 或 timer 的觸發時間，接著再觸發 event 或 timer，跳去指定執行的 function。Scheduler 就是不斷地重複以上動作來達成 event 的執行與模擬時間的推進。這樣的模擬方法，模擬時間是根據處理的 event 來推進，是不規則一段段向前躍進的，這便是其稱之為 discrete event simulation 的原因。

上述是僅考慮到不需與 kernel 互動的情況，但是我們說過 NCTUns 的一大優點就是模擬能夠直接使用現實生活中的 TCP/IP protocol，所以接下來便討論如何處理 kernel event。

Kernel 在處理模擬世界的封包時，如果有需要參考到時間的部分，要參考的對象必須是模擬時間而不能是現實時間；例如在一模擬的 node 上執行 ping 程式，ping 程式預設一秒發送一個 ICMP 封包去探測，此處的一秒指的就必須是模擬時間的一秒，而非現實生活中的一秒。但是承上所述，模擬時間是由 user level 在維護，kernel level 該如何去拿到模擬時間這個資訊？

NCTUns 在此採用 memory map 的技術，使得 kernel 宣告一塊在 kernel space 的記憶體空間，而這塊記憶體空間能被在 user level 的 simulation engine 所操作。

當 kernel 在處理模擬相關事宜時使用的 timer，因為必須全部以 memory map 技術取得的模擬時間為標準，所以原本 kernel 處理 timer 的機制並不適用，kernel 必須想辦法讓 user level 的 simulation engine 來處理。

首先這些模擬時間的 timer 都會被放入一個由 NCTUns 產生的在 kernel 內的 virtual timer list，同樣根據觸發時間去排序，與此同時 kernel 會發出一個同樣觸發時間的 event 丟入 event tunnel 的佇列中。

Event tunnel 用以傳遞 kernel event 給 simulation engine，scheduler 會發現有 kernel event 傳來，就將它抓起來並丟進 event heap 內根據觸發時間排序。當 scheduler 因為模擬時間與這個 event 的觸發時間最近而把它取出時，模擬時間同樣推進到此 event 的觸發時間，並且會呼叫 NCTUns 新增的 system call 去通知 kernel，kernel 便會觸發 virtual timer list 中與此 event 的觸發時間相同的 timer。

如此達到了無論是 user level 或是 kernel level 產生的模擬 event 都統一由 scheduler 來處理並且依此推進模擬時間。

2.5 Process scheduling 設計

由於 NCTUns 能夠直接讓現實生活中的 Applications 加入模擬產生流量，因為 Applications 是一隻 process，simulation engine 也是一隻 process，在競爭 CPU 的使用權時，它們的地位是相等的，這樣的話就有可能會發生問題，因此在 process scheduling 方面需要作一些額外的設定才能夠使模擬順利運行。

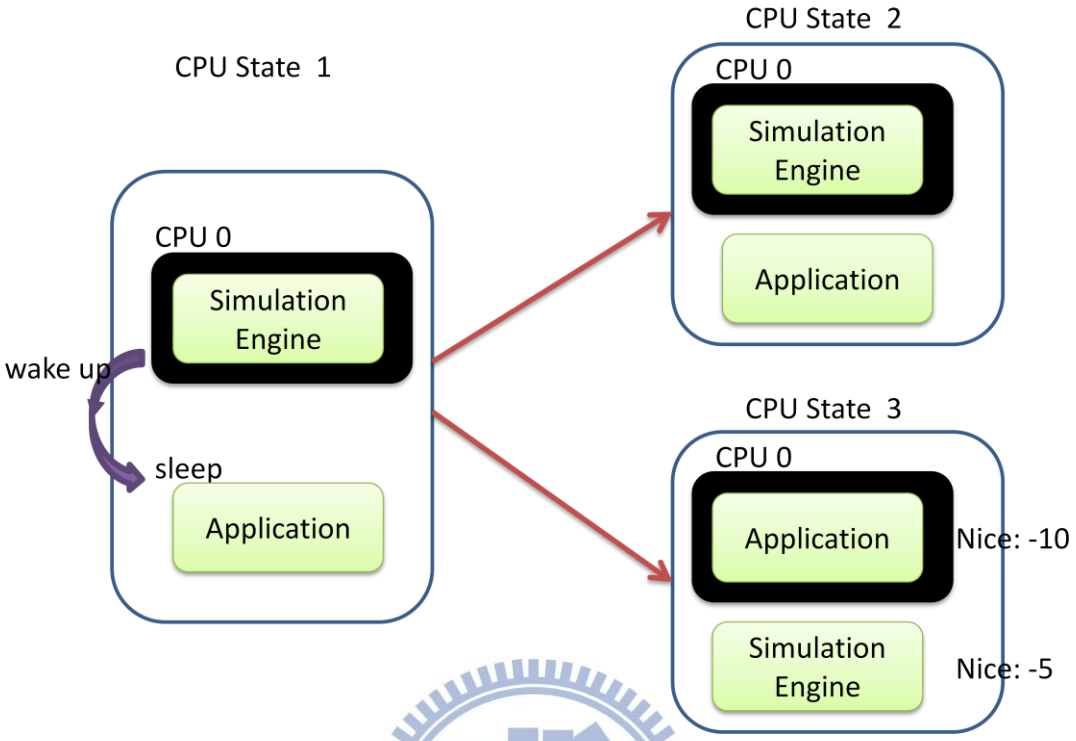


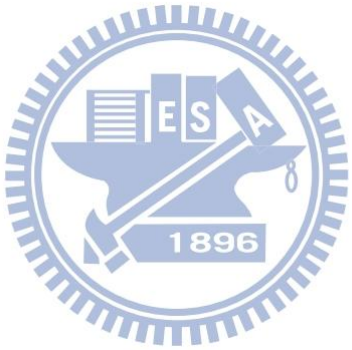
圖 2.6 Process scheduling problem

我們用上圖 2.6 來解釋，CPU state 是指 process 占用 CPU 的情況，如 state 1 就表示 simulation engine 正占用 CPU 執行，而有一隻 Application 正在 sleep；現在 simulation engine 喚醒 Application 執行，由於兩者地位相當都是要競爭 CPU 的 process，因此有可能進入到 state 2，Application 搶不到 CPU，由 Simulation engine 繼續佔用 CPU。但如同 2.4 節所提到的，simulation engine 會不斷由 event heap 或 timer list 中取出最近的 event 或 timer 處理並推進模擬時間，這導致原本在模擬時間 t 就應該被喚醒且執行的 Application 可能在模擬時間被推進為 t+n 後才得到 CPU 使用權，但此時 Application 送出封包的時間戳記就不再是 t 而是 t+n。這個現象會使得我們無法確信我們的模擬結果。

解決這個問題的方法就是為 Application 指定較高的執行優先度，如此在 simulation engine 一喚醒 Application 時，由於 Application 的執行優先度

較高，它就會搶占 CPU，使得 simulation engine 無法繼續推進時間。但是此處注意，被設定較高優先度的 Application，程式中必須要有像定期 sleep 這樣的機制去釋放 CPU，否則將會一直占用 CPU 的使用權，不僅模擬不能運行，連日常工作都會受到影響。

指定執行優先度這件事可以經由設定 process 的 nice 值做到，當 nice 值越小，表示該 process 的執行優先度就越高。我們見到當設定 nice 值後，圖 2.6 中 state 1 就必定進展至 state 3，也就不會有前述問題。



參、設計與實作

3.1 設計概觀

重新檢視一下我們的研究目標，我們想要讓 NCTUns 提供並行模擬，能夠在同一時間進行多項的模擬工作，並利用原本版本 NCTUns 運行在多核心系統下閒置的多顆 CPU，不浪費這些運算資源。

為了達到這個目標，設計上我們仍舊採用原本 dispatcher、coordinator、GUI Client 的分散式架構，一個 coordinator 代表一個 simulation engine 向 dispatcher 註冊，dispatcher 根據 GUI 的請求分配閒置的 coordinator 提供模擬服務，只是現在經過我們的修改，多個 simulation engine 可以並存於同一台 Host 裡，並且該 Host 支援 SMP 架構。如圖 3.1 在 Host 1 上同時就有 3 隻 coordinator 執行在 Host1 上，當三隻 coordinator 任一收到模擬請求，都會 fork 出自己的 simulation engine 來進行模擬；僅管在同一台主機上，但這些 simulation engine 之間是彼此獨立，並不會互相影響。

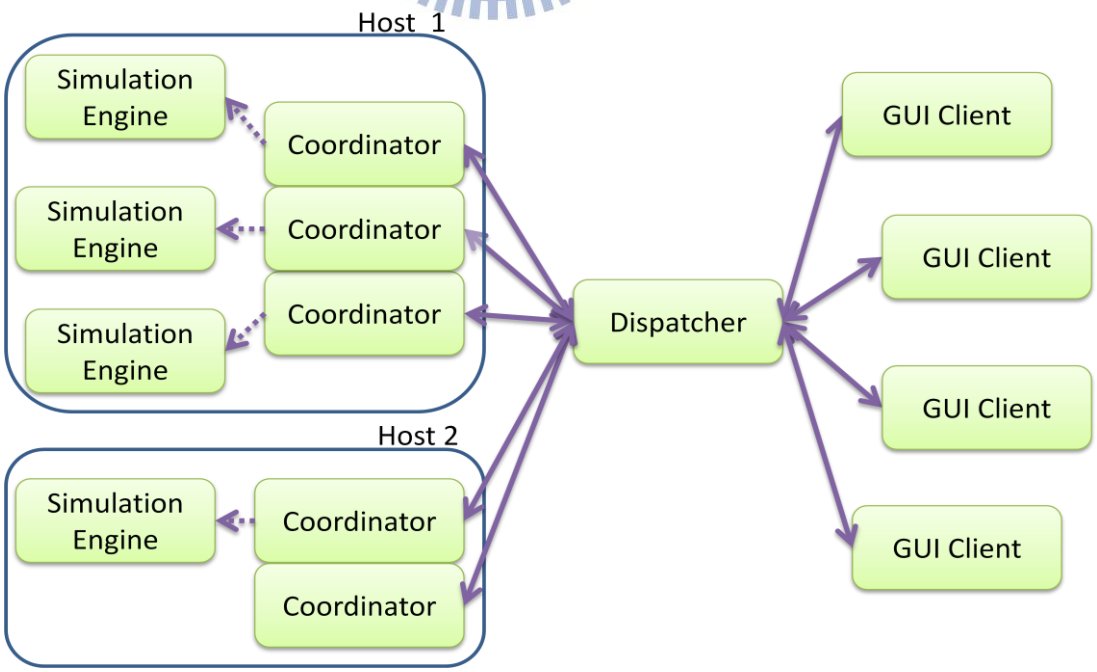


圖 3.1 NCTUns 新架構

當然也可以像原始版本的 NCTUns，在一台 Host 上只執行一隻 coordinator 與 simulation engine，由僅一隻的 simulation engine 程式來負責多項模擬工作的執行，但一來這樣的設計實作將會變得十分複雜，像是多項模擬過程間的獨立性等等，再者考慮安全性的問題，若是在模擬過程中程式崩潰，只有一隻 simulation engine 的設計可能就導致全部的模擬任務失敗，而我們提出的設計可能就只有一項模擬工作失敗，其餘的模擬工作還能繼續。

要實現這樣的設計我們整理後主要有三項修改。

1. 綁定 CPU
2. 修改系統核心
3. 修改模擬設定檔案

這三點將在下面的小節詳加介紹。

3.2 綁定 CPU

3.2.1 多核心系統下遭遇的問題

如同前面 2.5 節所敘述，NCTUns 利用指定 simulation engine 與執行在其上的應用程式的 nice 值，令應用程式的執行優先度高於 simulation engine，達到保證模擬順序的目標。

這個方法在單核心系統下能夠成功的達到目的，但在多核心系統下，即便應用程式的執行優先度高於 simulation engine，仍有可能發生應用程式與 simulation engine 分別同時執行在不同的 CPU 上如圖 3.2，這種情況會導致模擬時間的推進錯誤，因為執行 simulation engine 的 CPU 沒有被應用程式搶佔，使得 simulation engine 會持續處理新的 event，並將模擬時間推進至該 event 的模擬時間，則此時應用程式所產生的 event，它的時間戳記就會是新的模

擬時間，而非它原本該是的模擬時間。因此在多核心的系統下，原本的 NCTUns 是無法進行正確模擬的。

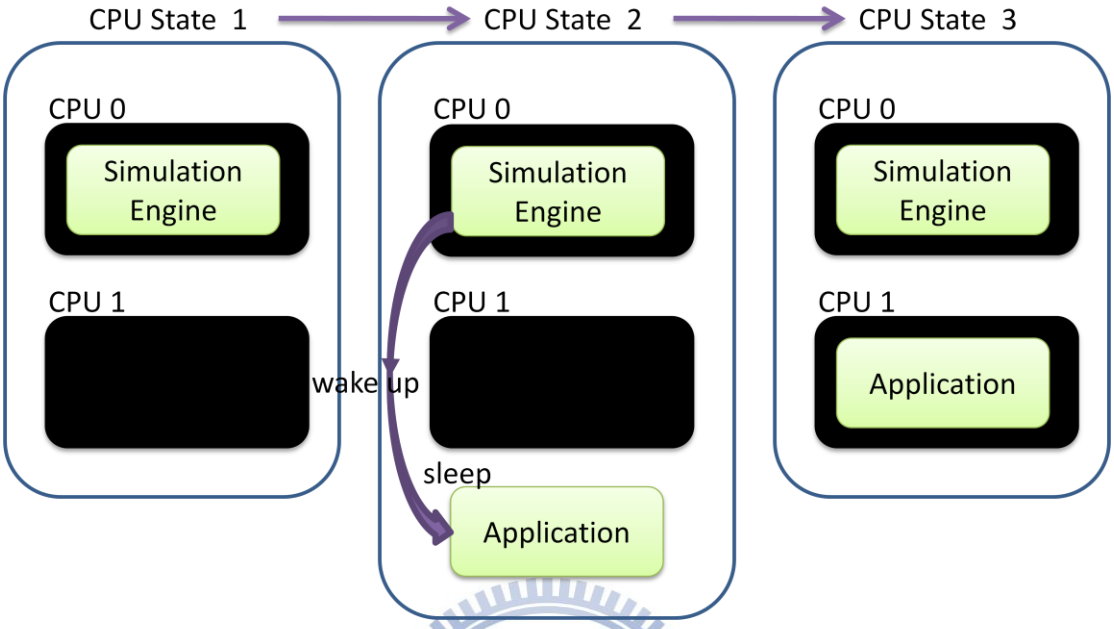


圖 3.2 多核心系統下的問題一

3.2.2 CPU 親和力

上一小節提到的問題，發生的主因是原本單核心系統下，同一時間只能有一隻程式能取得 CPU 的使用權，因此 simulation engine 的模擬時間推進必然發生在全部的應用程式處理完它在該模擬時間該做的工作之後，而在多核心系統下卻能有多隻程式同時取得 CPU 的使用權。在此我們將使用 CPU 親和力的概念來解決這個問題。

所謂 CPU 親和力是指在 SMP 架構下，能夠將指定的行程綁定在一個或多個特定的 CPU 上執行，原本是用於避免行程的執行效能降低，因為經由排程一隻行程可能會持續在不同 CPU 間切換，這樣的情況會造成不斷的 CPU cache miss，使得效能降低。因此發展 CPU 親和力的概念將行程綁在固定 CPU 上執行，才能徹底利用 CPU cache。

Linux 提供了 taskset 命令來操作 CPU 親和力，當我們想指定特定行程只能在特定 CPU 上執行時可以利用：

taskset –p [mask] [pid]

這樣的命令輸入，pid 欄位填入欲指定的行程的 process ID，mask 欄位為 CPU 遮罩。所謂 CPU 遮罩是用以表示 CPU 間的組合，在此即指出要綁定的 CPU 組合。舉例來說現在有一台 32 核的機器，當 CPU 遮罩為 0x00000001 即表示指定第 0 顆 CPU，若為 0x00000003 則是表示指定第 0 顆和第 1 顆 CPU，因為將其轉為二進位最右兩 bit 為 1。

如果要在 C 語言程式碼中去指定行程綁定的 CPU，需要引入<sched.h>標頭檔，再使用：

int sched_setaffinity(pid_t pid, unsigned int cpusetsize, cpu_set_t *mask)

第一個參數是要綁定的 process ID，當值為 0 表示是當前執行這段程式碼的 process，第二個參數是 CPU 遮罩的大小，第三個參數是 CPU 遮罩。若是設定成功則回傳 0，失敗回傳 -1。

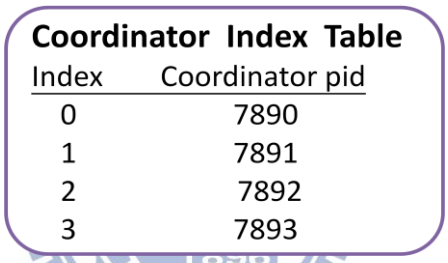
而 child process 的 CPU 親和力屬性會繼承自 parent process，所以經由上述兩種方法設定了 CPU 親和力後，該行程 fork 出的 child process 也會綁定在同樣的 CPU 上，這點對我們而言是很方便的，因為當我們綁定了 simulation engine，在模擬過程中所需要的 Application 其實皆是由 simulation engine fork 出來，如此它們自然也會綁定在同一顆 CPU 上。

如果我們能夠將 simulation engine 與其所 fork 出的 Application 全部綁定在某特定 CPU 上，那麼對它們而言就是在競爭同一顆 CPU，當執行優先度較高的 Application 執行，simulation engine 將被中斷，不會再有模擬時間錯亂的問題。並且藉由將不同套的 simulation engine 綁定在不同 CPU 上也可以使用到原本閒置的 CPU，提高 CPU 的使用率。

3.2.3 coordinator index 機制

承上節，對於每一套 simulation engine 我們知道可以經由 CPU 親和力的方法為其指定特定的 CPU 執行，但是 simulation engine 該如何決定自己要綁定哪一顆 CPU 執行？為了彈性考量，我們希望 simulation engine 在執行的時候會自動去決定自己該綁定的 CPU index，所以我們建立一套機制來做這件事情。

我們將建立一張表格稱為 coordinator index table，儲存 coordinator 的 process id 與分配給它的 index，型式如圖 3.3。像圖 3.3 代表的意義就是此時系統同時執行了 4 個 coordinator。



Coordinator Index Table

Index	Coordinator pid
0	7890
1	7891
2	7892
3	7893

圖 3.3 coordinator index table

這張表格會由 coordinator 來操作，當一隻 coordinator 開始執行時它會先去檢查這張表格是否存在，若是不存在便建立並將自己填入表格內容，若是表格已經存在，則 coordinator 會由上而下遍歷整張 table，找尋空的欄位並填入自己的 process ID；而當 coordinator 結束時會將自己的欄位由這張表格刪除。

簡單的操作示意如圖 3.4，原本同時有 4 隻 coordinator 在執行，之後 process id 為 7891 和 7893 的兩隻 coordinator 被關閉，表格內相應的 process id 欄位就會被改為 -1，表示該 index 尚無 coordinator 占用，最後又一隻 process id 為 7895 的 coordinator 被執行，則這隻 coordinator 由首至尾搜尋發現 index 1 的欄位為空，便將自身 process id 填入此欄位，結束操作。

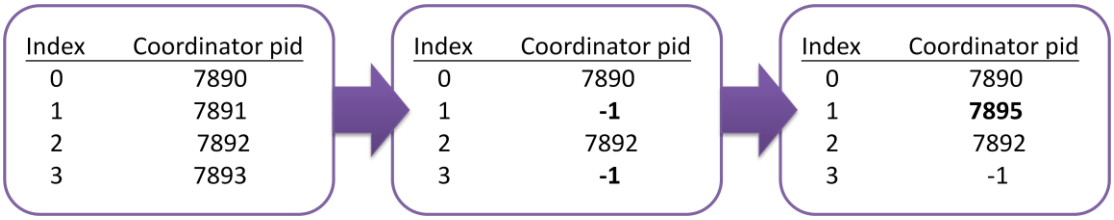


圖 3.4 coordinator index table 操作示意

建立這張表格的用處在於同時執行的多套 simulation engine 能夠藉此來分辨彼此，因為 coordinator 與 simulation engine 是一對一的關係，不會同時存在由同一隻 coordinator fork 出的兩套 simulation engine。而 simulation engine 能夠藉由查詢自己的 parent process 也就是 coordinator 的 process id 來取得獨一無二的 index。

既然目的是要分辨 simulation engine，那麼為什麼在這邊我們不選用 simulation engine 的 process id 來作為決定 index 的標準？NCTUns 裡 coordinator 每進行一項模擬工作就會 fork 出一隻 simulation engine 來處理，當模擬結束該隻 simulation engine 也結束，所以若是以 simulation engine 的 process id 來決定 index，需要頻繁的讀寫這張表格，是不必要的浪費。

當取得 coordinator index 後，simulation engine 就藉此來決定想綁定的 CPU index，當然不是直接套用，試想在四核心 CPU 的系統下，同時執行 5 隻 coordinator，那 coordinator index 為 4 的 simulation engine 想綁定第 5 顆 CPU 就會發生錯誤，因此欲綁定的 CPU index 就設定為 coordinator index 對 CPU 的數量取餘數。

以下是節錄自 coordinator 初始化時的一段程式碼：

```
/*  at  src/coordinator/init.cc  */
Int  CpuNum  =  sysconf(_SC_NPROCESSORS_CONF);
cpu_set_t  mask;
CPU_ZERO(&mask);
CPU_SET(CoorIndex%CpuNum, &mask);
if(  sched_setaffinity(0,  sizeof(mask),  &mask)  ==  -1  )
      printf("Set  CPU  affinity  fail\n");
```

首先宣告一變數 CpuNum 取得系統 CPU 的數量，再宣告一 CPU 遮罩 mask，並以 CPU_ZERO 宏對 mask 初始化，再以 CPU_SET 設定 mask 要指定的 CPU index 為 Coordinator index 對 CPU 數量取餘數。最後再利用 sched_setaffinity()與 mask 設定 CPU 親和力。

上述要注意的是呼叫 sched_setaffinity()函數時，我們傳入指定的 process id 是 0，意指要綁定的行程是 coordinator 自己，為什麼我們在意的是 simulation engine 和它所 fork 的應用程式需要綁定 CPU 執行，但卻在 coordinator 處就先綁定 CPU，而不是在 simulation engine 的程式碼內綁定自己。這是因為我們無法確定呼叫 sched_setaffinity()後，系統排程器會需要多久才能將行程遷移並綁定在指定的 CPU 上，如果在 simulation engine 處才設定 CPU 親和力，要是發生等到 simulation engine 開始 fork 出應用程式都還沒遷移完成的情況，可能就會造成錯誤，所以我們為了給予一點緩衝時間，在 coordinator 初始化時就進行設定，由於 simulation engine 會繼承 coordinator 的 CPU 親和力，在產生之初便已綁定在指定的 CPU 上。

3.3 修改系統核心

3.3.1 多核心系統下遭遇的問題

當我們在多核心系統上同時執行多套的 simulation engine，由於是分別獨立執行，屬於 user level 的 simulation engine 的記憶體空間彼此獨立，並不會造成衝突。但因為 NCTUns 還需要與 kernel 互動，我們雖然執行了多套 NCTUns，但 kernel 部分卻是共享的，所以對於 kernel 內部需要再做修改。

用下圖 3.5 來說明這個問題，原本 NCTUns 的模擬時間是由 simulation engine 來推進，並且以記憶體映射(memory map)的方式儲存在 kernel 內，當

kernel 只要有與時間有關係的操作，都需要取用這份模擬時間作為依據。假設現在執行兩套 NCTUns 進行模擬工作，可能第一套 NCTUns 模擬至 100 秒，而第二套 NCTUns 才開始模擬，則由於系統核心內的模擬時間只存有一份，無論是保持原本的 100 秒，或被覆蓋為 0 秒，都會造成模擬的錯誤。

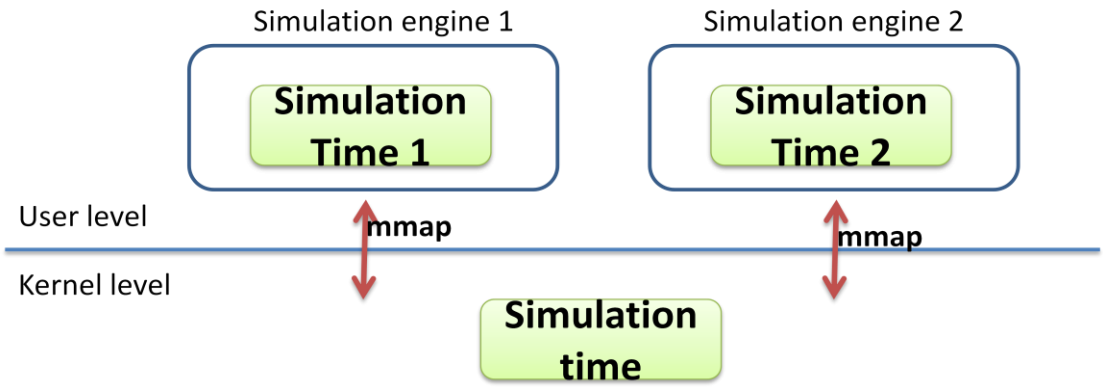


圖 3.5 多核心系統下的問題二

這類型的錯誤都是因為在原本版本的 NCTUns kernel 裡，所有為模擬所做的修改，都只有考慮到一份模擬引擎，如果為每一份參與並行模擬的 simulation engine 都準備一份專屬於它的 kernel 模擬機制，這個問題就得以解決。我們以圖 3.6 來說明如何解決圖 3.5 的問題，當我們在 kernel 裡也準備兩份 memory map 的空間分開儲存模擬時間，這樣就能避免因多個 simulation engine 想與 kernel 互動時，kernel 的應對機制卻只有一套帶來的錯誤。

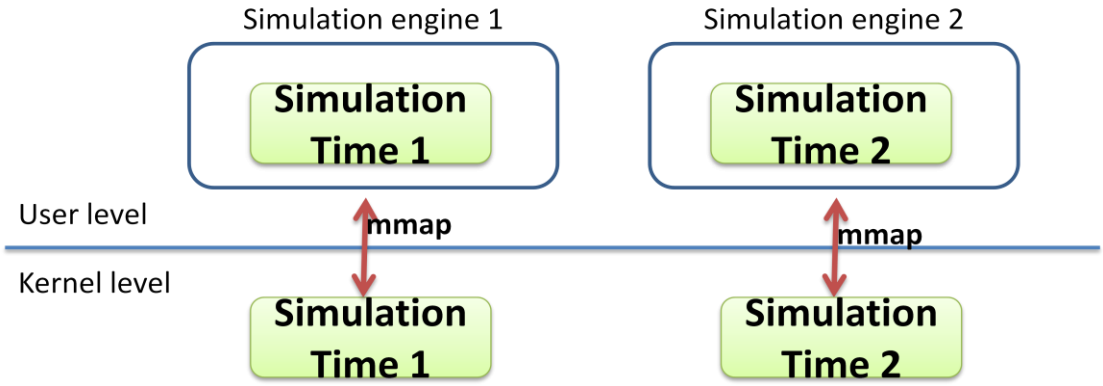


圖 3.6 多核心系統下的解決方法二

3.3.2 kernel 分辨 simulation engine

當我們已經作到複製多份 kernel 裡與 simulation engine 互動的機制，例如將原來單一個變數宣告為並行模擬數量大小的陣列，還存在一個問題；就是對於 kernel 來說，它如何去分辨今天來要求模擬機制的是哪一套 simulation engine，如果沒有辦法分辨，即使複製多份 kernel 裡的模擬機制，也無法順利模擬。

這問題的解決方法便是上節解釋的 coordinator index，但是如果 kernel 每次要作 simulation engine 判斷時都需要比對 coordinator index table 與現在執行中的 parent process ID，非常沒有效率，所以我們在 linux 描述 process 的資料結構 task_struct 裡新增了 CoorIndex 變數如下圖 3.7。

我們可以看到圖 3.7 左半部是原始版本 NCTUns 已做的修改，加入 nodeID 與 endtime。但 nodeID 僅能分辨該行程是否屬於 NCTUns，當執行多套 NCTUns 時 node ID 無法分辨該行程是屬於哪一套 NCTUns。因此我們在此新增一變數 CoorIndex，目的是讓 kernel 能夠在處理 NCTUns 相關服務時，能夠利用以下方式得到該去以哪一套模擬機制應對的資訊。Current 就是指向現正執行的 process。

```
current->nctuns_task.CoorIndex
```

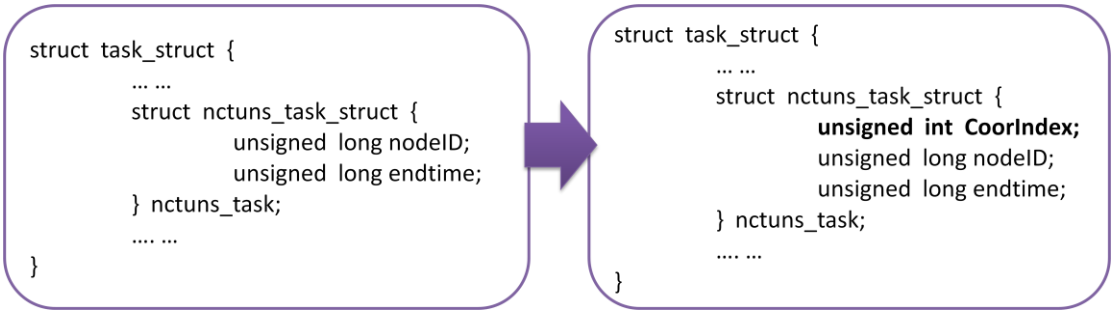


圖 3.7 linux process 資料結構

```
/*   at   src/patched_kernel/kernel/nctuns/nctuns_syscall.c   */

asmlinkage int sys_NCTUNS_misc(enum syscall_NSC_misc_enum action,
      unsigned long value1, unsigned long value2, unsigned long value3)
{
switch (action) {
case syscall_NSC_misc_SET_COORINDEX:
                        /*
                          * value1 => process ID
                          * value2 => CoorIndex
                          */
                        ptsk = find_task_by_pid_ns(value1, &init_pid_ns);
                        if(ptsk) {
                                ptsk->nctuns_task.CoorIndex = value2;
                                error = 0;
                        } else
                                error = -EINVAL;
                        break;
… …
}
```

現在系統核心內的操作都有行程資料結構內的 coordinator index 作為依據，而最初行程內的 coordinator index 又是如何填入。在此是採用以我們自定義的 system call 由使用者層級的行程來呼叫填入 coordinator index。精確的說便是當 coordinator fork 出 simulation engine，當 simulation engine fork 出 Applications，我們都會呼叫這個 system call 來填入 CoorIndex。

```
syscall_NCTUNS_misc(syscall_NSC_misc_SET_COORINDEX,
                                getpid(), CoorIndex, 0);
```

當

確保了我們能夠在 kernel 複製多份模擬所需要的機制以及 kernel 能夠準確判斷在屬於哪一套 simulation engine 的 process 處理時該使用哪一套 kernel 裡的模擬機制，我們也就能夠確定模擬過程中 simulation engine 與 kernel 的互動能夠成功。

經過我們整理歸納之後，在 kernel 需要這樣複製多份的模擬機制有

1. 負責模擬時間的維護與 kernel 裡相對應模擬時間的 timer list 的 NCTUNS_nodeVC 與 callwheel。
2. 負責與 simulation engine 溝通的 event tunnel 與記錄 event queue length 的 t0eqlen。
3. 負責記錄模擬拓樸資訊的 Mtable。

由於概念都是一樣的，都是兩個部分，一是在宣告時將原有資料結構宣告為可並行模擬數量大小的陣列，二是在使用時注意該對應 coordinator index 去取得對應的資訊。在以下小節我們便以 NCTUNS_nodeVC 與 callwheel 作為範例來說明修改的細節。

3.3.3 範例：NCTUNS_nodeVC

NCTUNS_nodeVC 是 kernel 儲存 global virtual time 的變數，kernel 藉 memory map 技術與 simulation engine 同步取得模擬時間，原本只能記錄一組的模擬時間，我們將之宣告為一組模擬時間陣列，大小是我們定義的最大可並行的 coordinator 數量，裡面儲存每組 simulation engine 對應的模擬時間。在 user level simulation engine 也會根據自己的 coordinator index 更新相應的模擬時間。

```
/*   at   src/patched_kernel/drivers/char/nctuns_dev.c   */

#define   ConcunCoorNum 8
```

```
/*   at   src/patched_kernel/drivers/char/nctuns_dev.c   */

u64        NCTUNS_nodeVC[ConcunCoorNum];
```

而原本在 kernel 裡關於 NCTUNS_nodeVC 的使用都要經過修改，例如下面所附程式碼，當要取用 NCTUNS_nodeVC 時，都是利用 current->nctuns_task.CoorIndex 去取得 coordinator index 作為索引值得到相應的模擬時間。這麼做儘管 kernel 還是共享，但是就能根據行程內附的 coordinator index 來決定該取用哪一個模擬時間。

```
/*   at   src/patched_kernel/include/nctuns/nctuns_ticks.h   */

#define NCTUNS_ticks_to_ns(i) \
          (NCTUNS_nodeVC[current->nctuns_task.CoorIndex]
          * NSEC_PER_NCTUNS_TICKS)

#define NCTUNS_ticks_to_us(i) \
          div64_u64(NCTUNS_nodeVC[current->nctuns_task.CoorIndex],
                    NCTUNS_TICKS_PER_USEC)

#define NCTUNS_ticks_to_ms(i) \
          div64_u64(NCTUNS_nodeVC[current->nctuns_task.CoorIndex],
                    NCTUNS_TICKS_PER_MSEC)

#define NCTUNS_ticks_to_sec(i) \
          div64_u64(NCTUNS_nodeVC[current->nctuns_task.CoorIndex],
                    NCTUNS_TICKS_PER_SEC)
```

3.3.4 範例：callwheel

callwheel 是由 NCTUns 自行建立在 kernel 裡的 virtual timer list，同樣如上面小節敘述的 NCTUNS_nodeVC，原先是只負責單一 simulation engine 的運行，為了支援並行模擬，在這也將 callwheel 宣告為可支援並行數量大小的陣列。

在 kernel 內的程式碼與上一小節提述的 NCTUNS_nodeVC 一樣，只要是有使用到 callwheel 的地方，都要修改為

```
callwheel[current->nctuns_task.CoorIndex]
```

如同以下所附程式碼展示新增 timer 進入 callwheel 的部分，只要有關 callwheel，全都以 coordinator index 去做存取的索引值。

```
/*    at    src/patched_kernel/include/nctuns/nctuns_callout.h    */
/*    at    src/patched_kernel/kernel/nctuns/nctuns_callout.c     */

 #define                callwheelsize      8192
uct callwheel {
              struct list_head vec[callwheelsize];
              int count;
 };
 struct callwheel              callwheel[ConcunCoorNum];
```

```
/*    at    src/patched_kernel/kernel/nctuns/nctuns_callout.c     */

/*    Insert timer into callwheel    */
int CoorIndex = current->nctuns_task.CoorIndex;
vec = callwheel[CoorIndex].vec + (expires & callwheelmask);
list_add(&timer->entry, vec);
(callwheel[CoorIndex].count)++;

nctuns_proc = find_task_by_pid_ns(nctuns, &init_pid_ns);
if(nctuns_proc != NULL)
       wake_up_process(nctuns_proc);
```

3.4 修改模擬設定檔案

3.4.1 多核心系統下遭遇的問題

當使用者利用 NCTUns 提供的 GUI 程式佈建模擬網路拓樸，設定模擬參數，這些設定都會由 GUI 產生許多設定檔來描述使用者的這個模擬 case，而為了方便使用者操作，GUI 也會 幫助 simulation engine 先做一些前期設定，例如最重要的一點就是自動產生 IP address，GUI 讀入使用者的拓樸後會自動根據區網的分割依序給予 Interface 們 IP，如果我們不去做任何修改，GUI 在分配 IP 與 tunnel ID 時都會從頭分配，每一個模擬 case 的 IP 與 tunnel ID 就很有可能重複到。

當如圖 3.8 發生 tunnel ID 重複的狀況，不同 case 的封包會寫入同一個 tunnel interface 且也由同一個 tunnel interface 被讀出，則不同 case 的封包會混淆在一起，造成模擬錯誤。

圖 3.8 tunnel ID 重複範例

如果 tunnel ID 沒有重覆，當 IP 重複也有可能發生問題，因為 NCTUns 在模擬網路時，是直接使用 kernel 裡的 TCP/IP protocol stack，routing 時也是直接使用原本的 routing 機制，以下圖 3.9 作為例子，模擬 case 一以兩台模擬主機互相以 tun1 tun2 interface 相連，他們分別被分配 IP 1.0.1.1 與

1.0.1.2，如此 simulation engine 會在模擬開始時加入 routing entry 到系統的 routing table，注意 node 下所列的 routing entry 已經是經過 2.3 節所述 S.S.D.D 的轉換，再看模擬 case 二，同樣是被分配 IP 1.0.1.1 與 1.0.1.2，而 tunnel ID 沒有重複，下方的 routing entry 明顯就與 case 一的 routing entry 相衝。這是由於兩項模擬在 routing 時都是使用系統裡同一張的 routing table，IP 的重複就造成 kernel 無法判斷該將封包送到哪一個 interface。

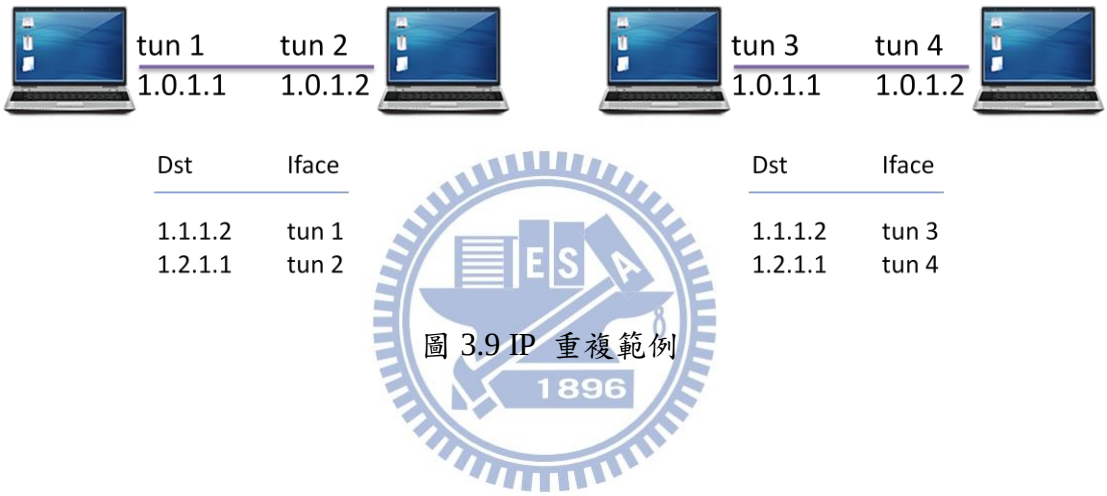


圖 3.9 IP 重複範例

3.4.2 remap tunnel ID and IP

如上節所述的問題，我們必須避開重複 tunnel id 與 IP address 的情況，我們決定使用 remap 的技術，將原本的 tunnel id 與 IP address map 到另外一個數值，這麼做對 kernel 來說，處理的等於是完全不一樣的情況，不會再有衝突，就像是圖 3.10 所示。

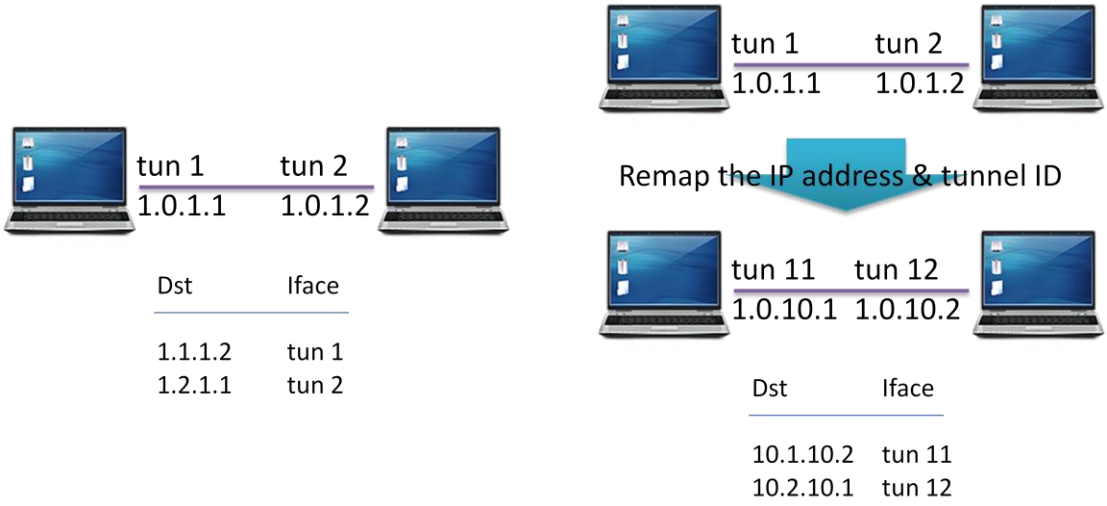


圖 3.10 remap tunnel ID 與 IP

由於我們是沿用原本 NCTUns 分散式的架構，對可能分布於世界各地的 GUI client 來說，它並不在意經由 dispatcher 分配給它的 coordinator 是否與其餘 coordinator 一同執行在同一台主機上，他只負責根據使用者的設定送出模擬的設定檔；要求 GUI 在產生設定檔時去根據 coordinator 的狀態去 remap tunnel ID 或 IP address 是不符合分散式架構概念且需要諸多改動的。因此我們決定不去更改 GUI 所產生的設定檔，而是在 simulation engine 讀入設定檔時再進行 remap，如此同一份設定檔就能夠在不同的 coordinator 環境下使用而不需修改。

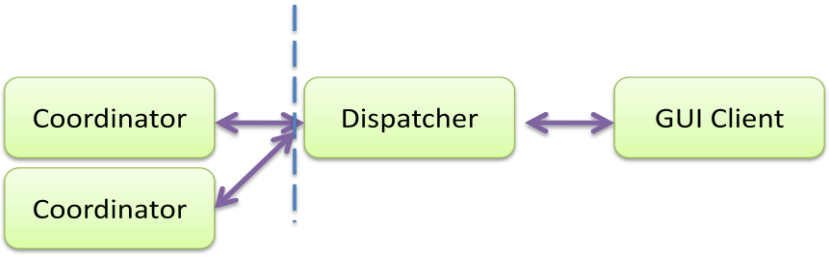


圖 3.11 NCTUns 的分散式架構

本論文 remap 的方式採取靜態分配，將 IP 的 subnet 與 tunnel ID 數量平均分配給所有參與的 coordinator。由於 NCTUns 的 routing 機制，IP 的前兩

位必須固定為 1.0，為了顧及同 subnet 下能分配的 IP 數量，決定以第三位作為分配的標準，也就是以並行 simulation engine 的數量去平分 256 個可以容納 256 個 host 的 subnet。

在一般的有線無線模擬 case 裡，只要是設定檔裡有包含 IP address 或 tunnel id 我們都需要去修改。舉一般的 ethernet 與 802.11a 802.11b 的模擬 case 作為例子，就需要在讀入以下三個檔案時進行 remap。

這三個檔案分別是.tcl 檔，.srt-l 檔，以及.tfc 檔，這是最基本的三個檔案，是無論進行什麼模擬任務都一定要 remap 的。其中.tcl 檔是描述整個模擬網路拓樸，內含 IP 資訊;.srt-l 檔是描述模擬任務需要添加的 routing entry，內含 IP 與 tunnel ID 資訊；.tcl 檔是描述模擬任務中執行的流量產生器，內含 IP 資訊。

由於都是 remap IP 與 tunnel ID，下面的小節我們以.srt-l 檔案的 remap 實作詳情作為範例來說明。

3.4.3 remap 範例：.srt-l 檔案

srt-l 檔案的內容是 GUI 根據使用者佈建的網路拓樸自動產生的 routing rule，型式可以分為兩種如圖所示，一種指定丟出的 interface，一種則是指定丟向 gateway。

```
route add -net 1.1.1.0/24 gw 1.1.4.2
route add -net 1.1.1.0/24 tun1
```

注意這裡的 IP 都是經過 S.S.D.D 轉換取用 source node 與 destination node 的 IP address 的第三第四 Byte 用以表示一條獨一無二的 routing rule，所以需要 remap 的就會是此處 IP 的第一與第三個 byte。

由於我們已經確定檔案內容的型式，於是先利用 strstr 函數確定是哪一種，接著可以直接使用 sscanf 抓出 IP 的資訊，並根據 simulation engine 自身

的 coordinator index 修改相應 byte 達到 remap 的目標。

```
while (RTFile.getline(line, 256))
{
    // ignore \n and #~
    if(strlen(line) > 0 && line[0] != '#')
    {
        iftun = strstr(line, "tun");

        if(iftun == NULL) // route add -net %d.%d.%d.%d/24 gw %d.%d.%d.%d
        {
            sscanf(line, "route add -net %d.%d.%d.%d/24 gw %d.%d.%d.%d"
                    , &ip[0], &ip[1], &ip[2], &ip[3], &ip[4], &ip[5], &ip[6], &ip[7]);

            ip[0] = ip[0] + MaxConcurSubnetNum*CoorIndex;
            ip[2] = ip[2] + MaxConcurSubnetNum*CoorIndex;
            ip[4] = ip[4] + MaxConcurSubnetNum*CoorIndex;
            ip[6] = ip[6] + MaxConcurSubnetNum*CoorIndex;

            sprintf(line_output, "route add -net %d.%d.%d.%d/24 gw %d.%d.%d.%d"
                    , ip[0], ip[1], ip[2], ip[3], ip[4], ip[5], ip[6], ip[7]);
            RTFile_M << line_output << endl;
        }
        else                        // route add -net %d.%d.%d.%d/24 tun%d
        {
            sscanf(line, "route add -net %d.%d.%d.%d/24 tun%d"
                    , &ip[0], &ip[1], &ip[2], &ip[3], &tunname);

            ip[0] = ip[0] + MaxConcurSubnetNum*CoorIndex;
            ip[2] = ip[2] + MaxConcurSubnetNum*CoorIndex;
            tunname = tunname + MaxConcurTunNum*CoorIndex;

            sprintf(line_output, "route add -net %d.%d.%d.%d/24 tun%d"
                    , ip[0], ip[1], ip[2], ip[3], tunname);
            RTFile_M << line_output << endl;
        }
    }
}
```

肆、效能分析

4.1 效能指標

為了比較原始版本與我們所修改的可並行模擬版本的 NCTUns，在這裡提出一個效能指標就是總模擬時間。 總模擬時間意指給予一定量的模擬任務，完成全部模擬任務所需耗費的時間。

為了讓讀者更清楚總模擬時間的定義，下面使用一個簡單例子說明。假設現在有兩個模擬任務 A 與 B 交付 NCTUns 執行，由於原始版本的 NCTUns 不支援並行模擬，它的總模擬時間就是任務 A 執行的時間加上任務 B 執行的時間；而對於可並行模擬版本的 NCTUns 來說，它的總模擬時間就是任務 A 執行的時間與任務 B 執行的時間中的最大值。

另外一個效能指標則是最大記憶體使用量，意指在模擬過程中，NCTUns 最大的瞬間記憶體使用量；我們想知道在使用並行模擬追求提高模擬總產出的同時，是否也要付出一些代價。所以在此觀察模擬途中所需要的記憶體使用量。

4.2 模擬環境

以下章節全部的模擬實驗都會在本節描述的模擬環境下執行。

CPU：AMD 6168　24 core 1.9G Hz

記憶體：4 GB

作業系統：Fedora 12

4.3 模擬實驗一

第一個實驗我們先測試最常見的 Ethernet 網路，藉由拓樸複雜度的增加，觀察在不同情況下，使用並行模擬的 NCTUns 與原本版本的 NCTUns 它們之間的效能指標會如何變化。

模擬參數如下表 4.1，注意變動的參數是模擬任務的拓樸 hops，所以下表表示的是一共有五組參數需要進行五次模擬，而每一次需要進行的模擬任務數量是 4，也就是第一次模擬的任務會是四個 2 hops 的網路模擬拓樸，第二次模擬的任務會是四個 3 hops 的網路模擬拓樸，依此類推。在我們接下來的模擬實驗，當需模擬任務數量這個參數若大於一，就是指多個其餘參數都一樣的模擬任務。

表 4.1 實驗一模擬參數表

模擬時間	**400 sec**
需模擬任務數量	**4**
模擬任務拓樸 hops	**2, 3, 4, 5, 6**
流量 protocol	**TCP**
流量數量	**1**

模擬拓樸如下圖 4.1，根據模擬參數 hops 的變化，在兩端以 Ethernet link 相連的模擬 node 間加入相應數量的 router。而無論 hops 的數量如何變化，最左端的模擬 node 都會建立一條 TCP greedy flow 流向最右端的模擬 node。

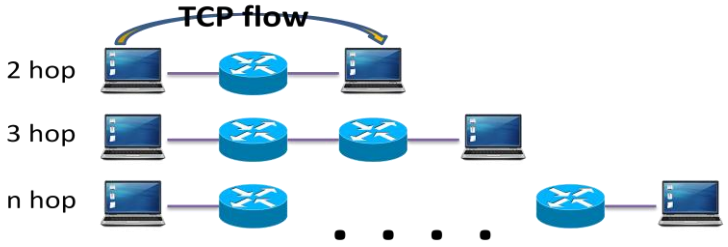


圖 4.1 實驗一模擬拓樸

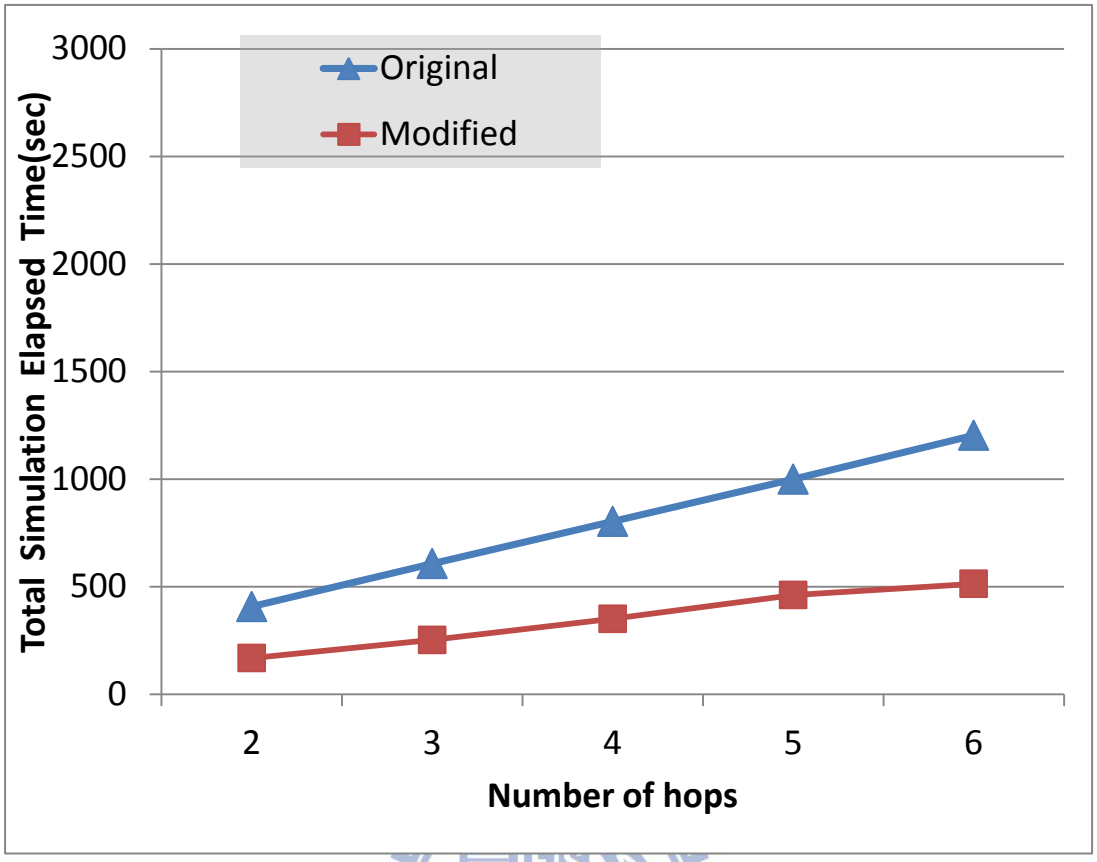


圖 4.2 實驗一總模擬時間

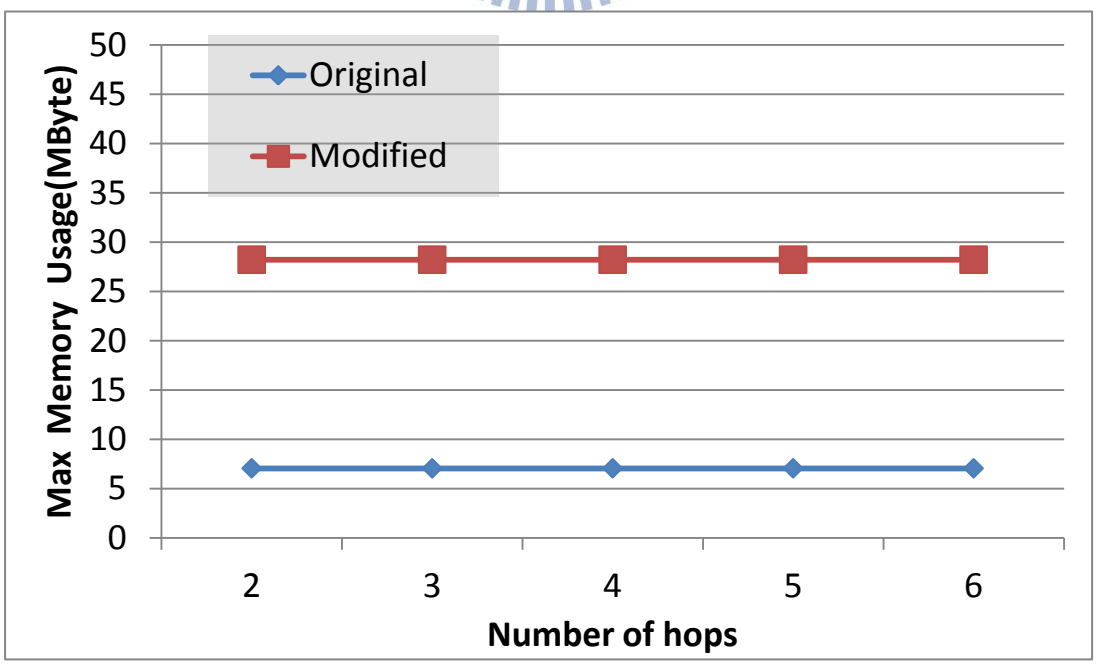


圖 4.3 實驗一最大記憶體使用量

觀察結果圖 4.2 可以發現即使隨著模擬拓樸 hop 數的增加，經我們修改過的支援並行模擬的 NCTUns 在總模擬時間上是穩穩優於原本的 NCTUns，提升的效能約略在兩倍與三倍之間。

另外觀察記憶體的使用量見圖 4.3，可以見到很明顯的當並行模擬四項模擬任務時，記憶體的使用量也是原來的四倍；不過我們可以發現在模擬 Ethernet 網路時原始 NCTUns 對於記憶體的需求並不高，因此即便因並行多項模擬使得記憶體需求倍數成長，也在可以負擔的範圍內。

4.4 模擬實驗二

第一個實驗我們已經測試可並行模擬的 NCTUns 在模擬 Ethernet 網路時的表現，第二個實驗我們將測試模擬無線網路的情況。

模擬參數如下表，在這裡需要注意模擬任務 node 數量與流量是對應的，2*2 的 node 數量對應到兩條 TCP 流量，3*3 的 node 數量對應到三條 TCP 流量，依此類推。因此實驗二一樣是僅有五組不同參數進行五次模擬。

表 4.2 實驗二模擬參數表

模擬時間	**400 sec**
需模擬任務數量	**4**
模擬任務 node 數量	**2*2, 3*3, 4*4, 5*5, 6*6**
流量 protocol	**TCP**
流量數量	**2, 3, 4, 5, 6**

模擬拓樸如下圖 4.4，根據模擬參數 node 數量的變化，佈下由 802.11b

ad-hoc mode mobile node 組成對應大小的格狀拓樸;而最左端直行的 node 都會傳輸 TCP 封包至最右端直行的 node 如圖所示。

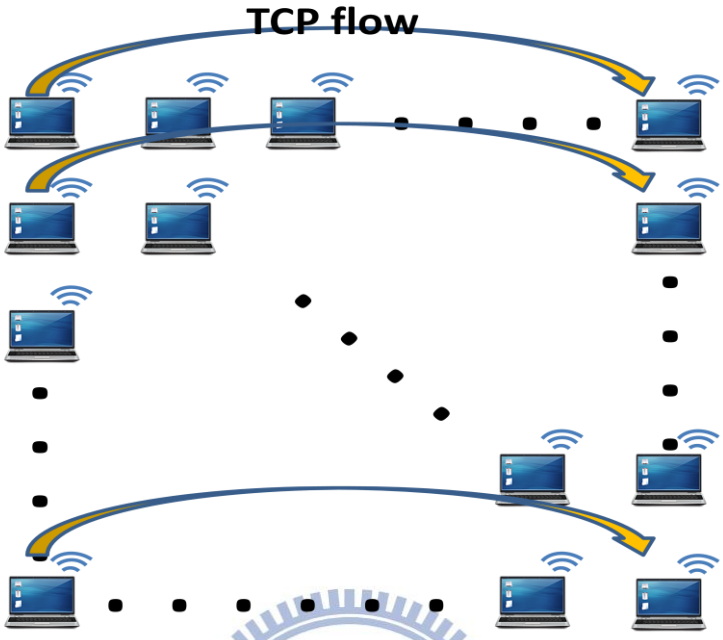


圖 4.4 實驗二模擬拓樸

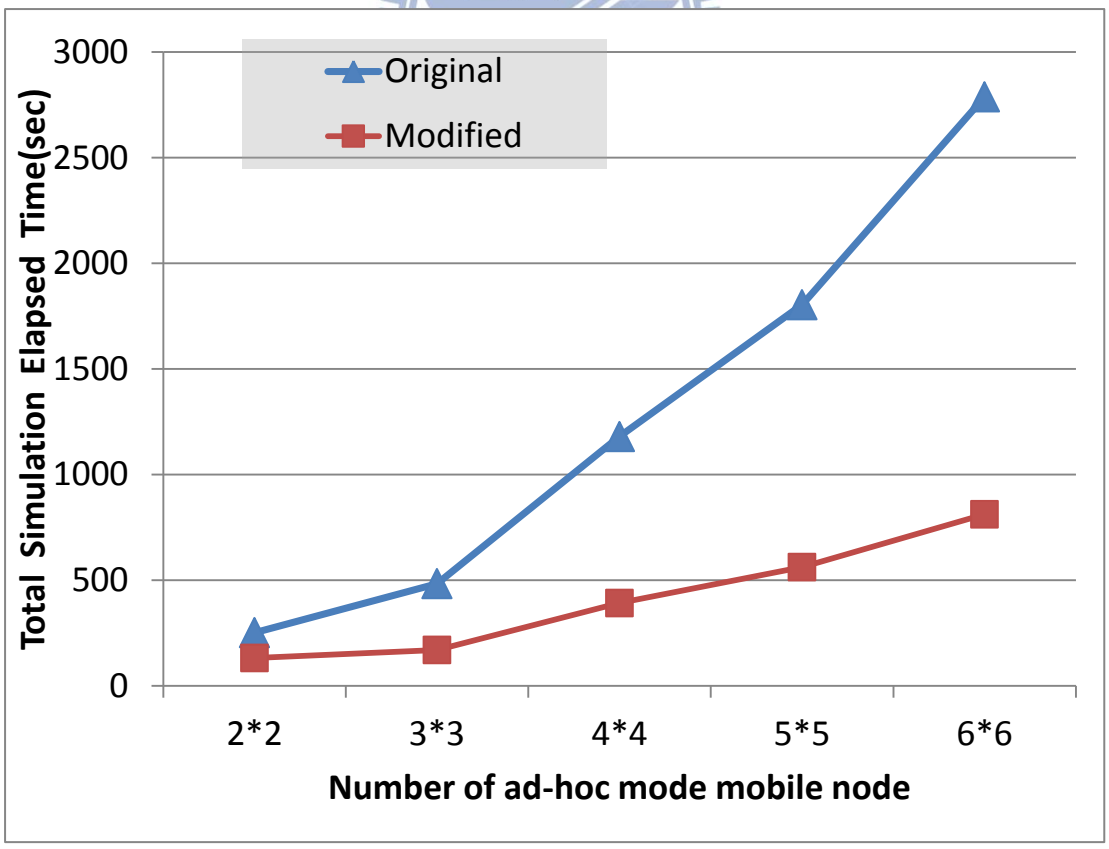


圖 4.5 實驗二總模擬時間

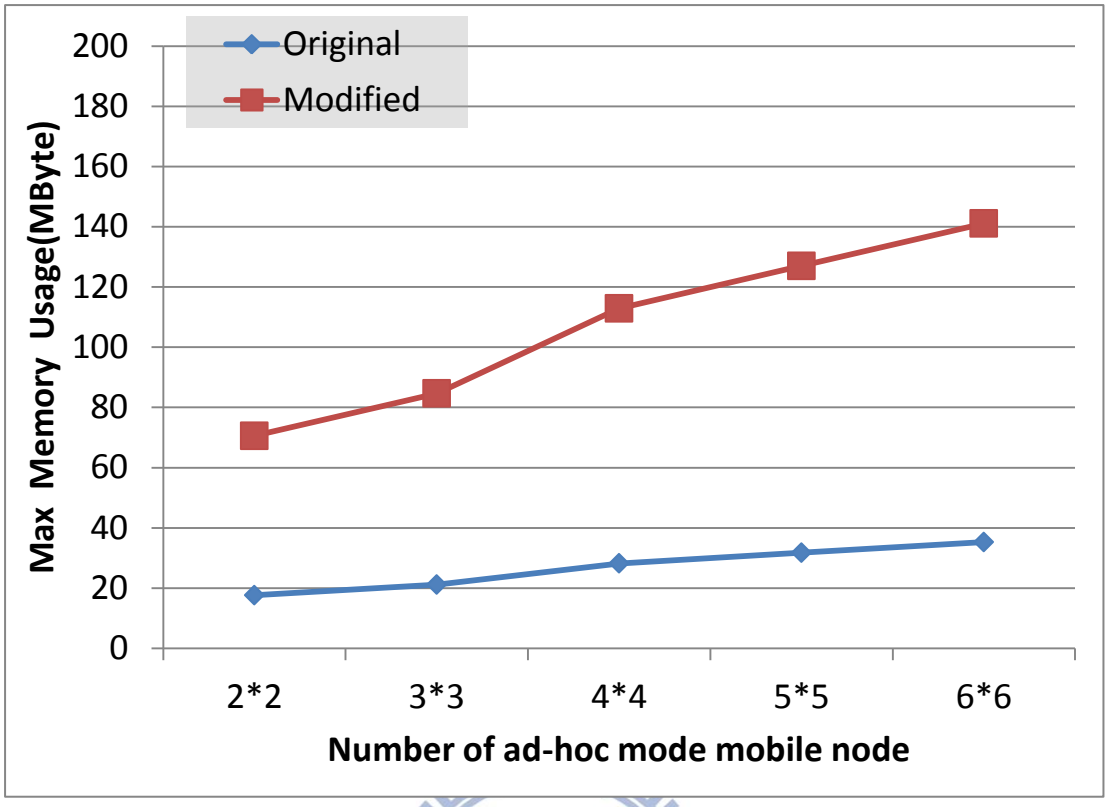


圖 4.6 實驗二最大記憶體使用量

觀察結果圖 4.5 發現可並行模擬的 NCTUns 在模擬無線網路方面，仍然表現十分優秀，總模擬時間遠低於原始版本的 NCTUns，提升的效能約略三倍。而關於記憶體的使用量，見圖 4.6 可並行模擬的 NCTUns 仍然需要耗費四倍的記憶體。

4.5 模擬實驗三

綜合前面兩個實驗結果，可並行模擬的 NCTUns 無論是模擬有線網路或是無線網路表現都非常的優異，在記憶體的花費上也因為原本 NCTUns 對記憶體的需求就不高，並不是太大的缺點。

但是實驗一與實驗二我們所設定的實驗參數裡，需模擬任務數量皆為四個，由於我們的 CPU 核心數量充足，模擬時就是四個模擬任務並行模擬；

今天如果並行模擬的數量提升，是否表現一樣優異？第三個實驗我們就來測試在不同的並行模擬數量下，對模擬的效能會有什麼樣的影響。

模擬參數如下表。

表 4.3 實驗三模擬參數表

模擬時間	**400 sec**
需模擬任務數量	**1, 2, 4, 6, 8**
模擬任務 node 數量	**4*4**
流量 protocol	**TCP**
流量數量	**4**

模擬拓樸如下圖 4.7，我們直接選用實驗二裡 4*4 方格的模擬任務作為比較環境。

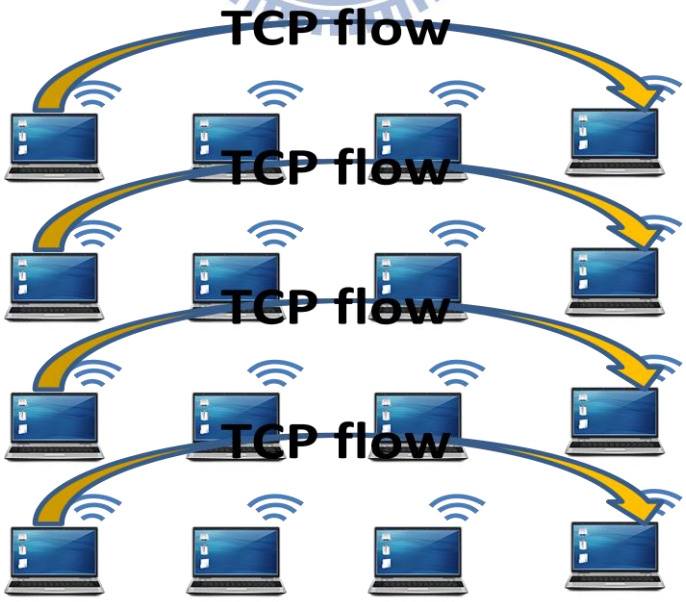


圖 4.7 實驗三模擬拓樸

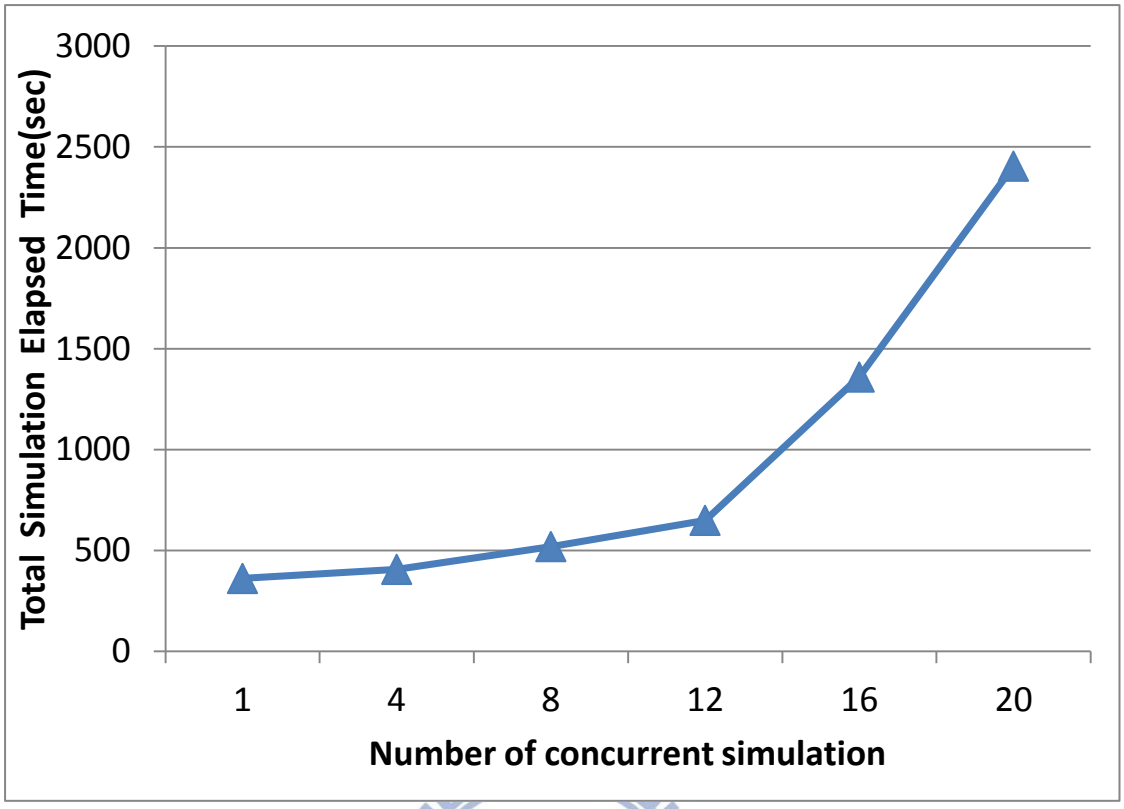


圖 4.8 實驗三總模擬時間

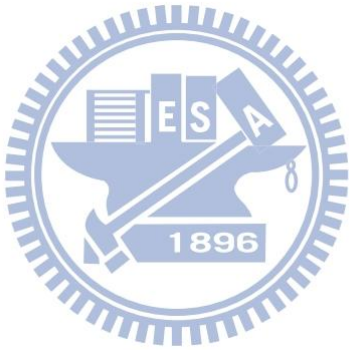
觀察圖 4.8，總模擬時間有指數上升的趨勢，在同時進行 20 項模擬工作時，總模擬時間接近 2500 秒；但注意如果是原本未支援並行模擬的 NCTUns，它執行完這 20 項模擬工作理論上便需要 6000 秒，明顯可見並行模擬還是在效率上占有極大優勢。

並行模擬的數量果然會造成模擬效能的降低，由於我們的 simulation engine 已經各自綁定 CPU，因此並不會是因為競爭 CPU cycle 導致效能降低。

在此提出兩個推測；

第一， 在 kernel 裡有些機制為了避免 race condition 是上鎖的，如果每套 simulation engine 都需要運行 kernel 裡這類型的機制，那麼由於一次只能服務一套 simulation engine，當並行模擬時就會拖慢整體的速度。

第二，　在 kernel 裡有些在模擬時被拿來共用的資料結構也可能造成拖慢情況，例如 routing table，由於全部的 simulation engine 都使用同一張 routing table，假設每個 simulation engine 所要加入的 routing entry 數量都十分龐大，那麼最終 kernel 裡這張 routing table 它的大小也會膨脹到十分驚人；此時每次進行 routing 時，由這張 routing table 搜尋出對應的 routing entry 的時間也會增加，造成模擬時間的拖長。



伍、討論

5.1 虛擬機器

回顧我們的研究動機與目的，由於原本 NCTUns 設計上的侷限，使得 NCTUns 只能在沒有支援 SMP 架構的 kernel 上執行，換句話說，當 NCTUns 執行在一台多核心的系統上，它只能使用一顆核心，且其餘的核心也不能用來執行作業系統其他工作，等於完全浪費。為了利用這些被浪費的 CPU 運算資源，於是我們決定修改 NCTUns，使它能夠正確地在 SMP kernel 下執行並且支援並行模擬，同時進行多項模擬工作。

能在 SMP kernel 下進行模擬並且支援並行模擬這樣的目標，其實除了我們提出的去修改 NCTUns 本身，我們會發現利用虛擬機器技術能夠更簡單的達到，想同時服務幾項模擬請求就創建幾台虛擬機器，而每台虛擬機器都已安裝 NCTUns，使用者可以在虛擬機器內進行模擬，以現在各家虛擬機器軟體皆提供友善的使用者介面的情況，可以說能與我們做到同樣的事而完全沒有技術門檻。

這一章我們就要比較使用我們的設計與使用虛擬機器兩種方式在各方面的優劣。虛擬機器我們選擇 Xen 與 VirtualBox 兩款虛擬機器軟體作為範本，這是由於該兩款虛擬機器是免費、知名並且容易取得的，下面的小節中倘若有關於虛擬機器的描述，發現有其餘版本的虛擬機器軟體可以做得更好，我們也相信這類軟體或者知名度不顯或者需要購買，對一般使用者來說不易接觸，因此本章還是僅拿 Xen 與 VirtualBox 與我們修改過的可並行模擬的 NCTUns 作比較。

5.2 總模擬時間

模擬參數如下表 5.1，基本上是沿襲上一章節實驗二的例子，以變化無線網路的模擬 node 數量，去觀察使用我們所修改的可並行模擬的 NCTUns 與兩種虛擬機器軟體，Xen 與 VirtualBox 的表現。

表 5.1 實驗四模擬參數表

模擬時間	**400 sec**
需模擬任務數量	**4**
模擬任務 **node 數量**	**2*2, 3*3, 4*4,** **5*5, 6*6**
流量 protocol	**TCP**
流量數量	**2, 3, 4, 5, 6**

模擬拓樸如下圖 5.1，同樣最左端直行的 node 會不斷送出 TCP 封包給最右端直行的 node。

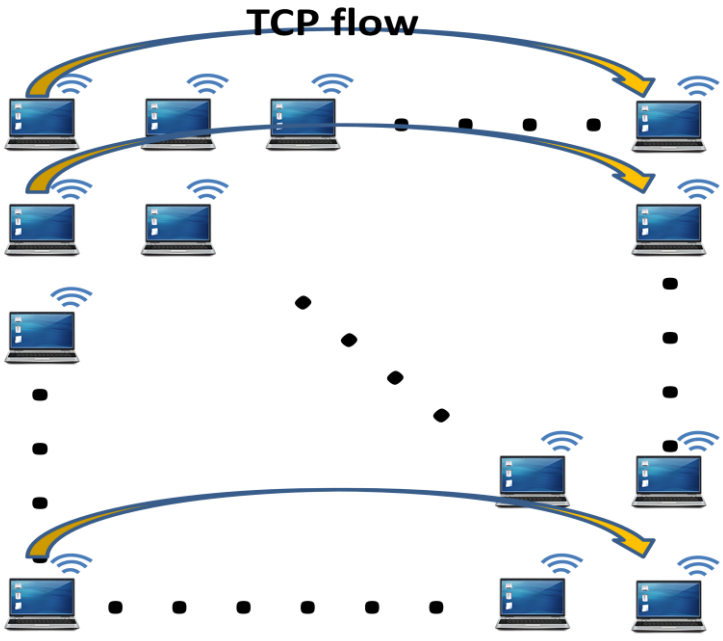


圖 5.1 實驗四模擬拓樸

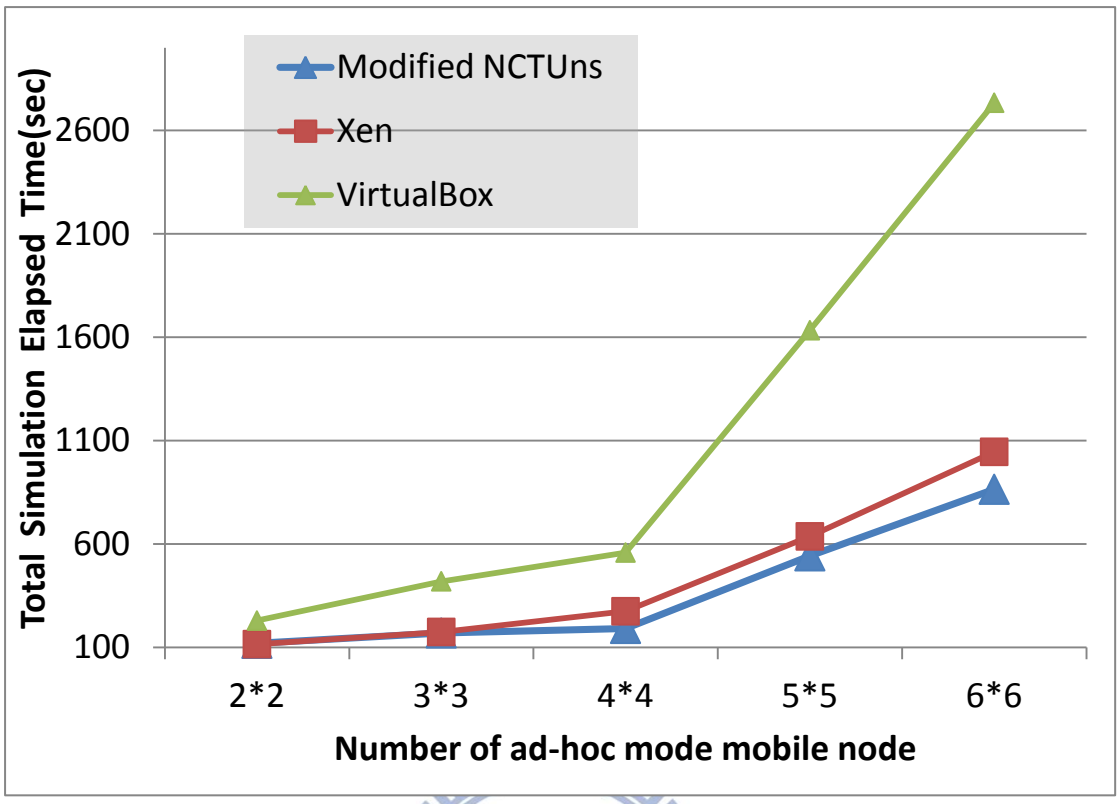


圖 5.2 實驗四總模擬時間

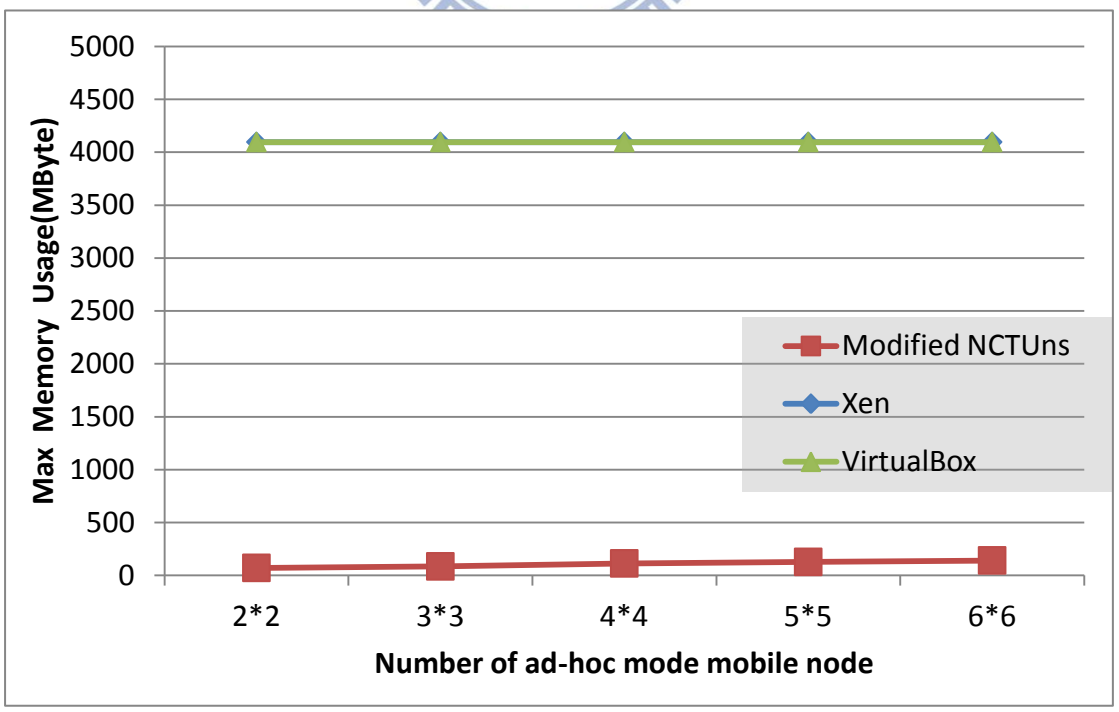


圖 5.3 實驗四最大記憶體使用量

由上面所述實驗結果圖 5.2 可以觀察到無論是 Xen 或是 VirtualBox 在模擬速度上還是要遜於可支援並行模擬的 NCTUns，並且穩定地呈現倍數的差距。這是因為虛擬機器畢竟不是真實機器，而是經過軟體令真實硬體為虛擬機器提供服務，等於多繞了一層。而 Xen 的表現會如此優於 VirtualBox 且貼近我們修改的可並行模擬的 NCTUns，推測是因為它使用 Paravirtualization (半虛擬化)，使得它對底層的硬體控制效率接近實機操作。但這樣的虛擬方式需要較高的硬體支援度，也需要作業系統的支援，可以說要求較為嚴苛。

最後見到圖 5.3，這張記憶體用量的圖很有趣，我們將它放在下一節比較硬體成本時討論。

5.3 硬體成本

在使用 Xen 或 VirtualBox 創建虛擬機器時或在每次啟動虛擬機器前，都必須決定該虛擬機器的記憶體大小，例如實驗四，我們為四台同時進行模擬的虛擬機器都分配了一 GB 的記憶體，觀看圖 5.3 就可以知道分配給虛擬機器的記憶體就無法再被實體機器所使用，等同於一台虛擬機器就完全吃掉了一 GB 的記憶體，相對於直接使用可並行模擬的 NCTUns 其記憶體使用量可說是非常微小。但若是我們降低分配給虛擬機器的記憶體大小，又有可能影響虛擬機器的運行，因為虛擬機器上還必須先安裝作業系統，分配給它的記憶體必須能保證作業系統能夠順利運行。

除了記憶體，耗費的硬碟空間也是值得探討的議題，直接使用可並行模擬的 NCTUns 所耗費的空間僅有 GUI 所產生的模擬設定檔以及模擬結束產生的記錄檔；但是使用虛擬機器，就必須再多耗費虛擬機器上作業系統的硬碟空間。

綜合以上，使用虛擬機器的方法所要求的硬體需求是較高的，當使用者

的機器沒有較高數量的記憶體與硬碟空間，使用可並行模擬的 NCTUns 是可兼顧模擬速度與硬體成本的選擇。

5.4 安全性

當這樣的兩套可支援並行模擬的方法，遭遇到不可預期的錯誤時，兩者對於錯誤的容忍度又是如何。

當一隻 simulation engine 無預警崩潰結束，使用虛擬機器方法的話，只會有該 simulation engine 負責的任務失敗，其它虛擬機器上 simulation engine 的任務並不會受到影響，仍然可以成功模擬。若是使用可並行模擬的 NCTUns，一隻 simulation engine 崩潰，也不會影響到其他的 simulation engine 工作，在這方面兩者表現可以說一樣好。

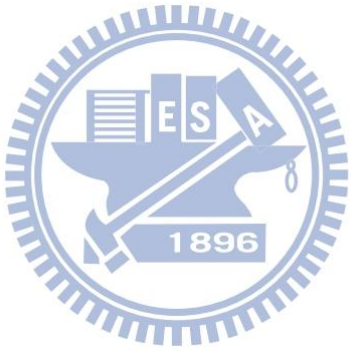
現在考慮是系統無預警崩潰結束的情況，使用虛擬機器方法的話，只會有該系統所在的虛擬機器需要重新開機，對於在其他虛擬機器上的模擬任務並不會造成影響；但如果是使用可並行模擬的 NCTUns，一旦系統崩潰，全部正在模擬的 simulation engine 也會同時崩潰。這一點是我們修改過的可支援並行模擬的 NCTUns 有所不足的地方。

5.5 管理性

在管理性上，虛擬機器也有不可比擬的優勢。當在一伺服器叢集裡虛擬機器可以很簡單地提供 migration 的功能，例如將一台模擬機器自一忙碌的伺服器移至一負載較輕的伺服器，達到負載平衡的功能。但可並行模擬的 NCTUns 就無法像虛擬機器一般藉由遷移虛擬機器來達到遷移模擬任務的效果。

5.6 擴充性

另外則是可擴充性的議題，今天是只單純討論提供模擬服務的情況，但日後如果提供模擬服務的伺服器想要提供新的功能給使用者，使用虛擬機器的話會十分方便，因為原本分配給使用者的就是一台虛擬機器，只要在虛擬機器上再新增提供服務的程式就好；但對於當時提供模擬服務是使用可並行模擬的 NCTUns 的伺服器，就必須再思考如何在新增的服務獨立使用者以便管理的問題。



陸、結論

我們根據 NCTUns 在利用多核心系統環境時的低效率，提出了開發支援並行模擬版本的NCTUns的目標，並且詳加探究NCTUns原本的設計架構、運行脈絡，整理出其在多核心系統環境下運行的三大問題，並針對問題一一提出解決方法，最後也成功實作出可並行模擬並充分利用多核心系統優勢的NCTUns。

比較我們的設計與原始版本的 NCTUns，可以發現在處理一定數量內的模擬工作時有非常明顯的效能提升；若是與虛擬機器比較，同樣能提供較好的效能，但對於較講求效率的虛擬機器軟體如 Xen，提升的效能就較為有限。然而虛擬機器所需耗費的硬體成本卻又是遠遠高於我們的設計，在硬體成本不足時，我們的設計會是進行並行模擬的最佳選擇。

柒、未來展望

7.1 動態 Remap

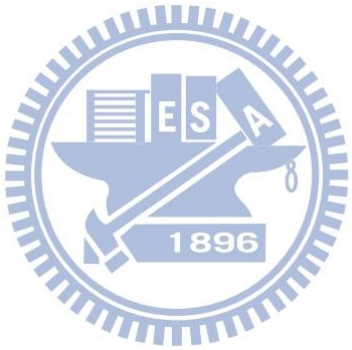
在本篇論文我們提到利用 remap 的技術去避開 IP address 與 tunnel ID 重複導致模擬無法成功運行的問題。我們使用的是靜態 remap，也就是根據並行模擬的 simulation engine 數量去平分 IP subnet 與 tunnel ID，這樣的平均分配會導致當所需 IP 數量大過分得的數量就無法進行模擬，而那些僅需要少數 IP 數量的模擬任務則是佔用了大部分的 IP。未來我們希望能夠依據模擬任務的網路拓樸動態分配需要的 IP 數量給該模擬任務就好，如此便能夠在資源真正飽和前盡可能提供模擬服務。Tunnel ID 的分配亦同此理。

7.2 自動 Remap

同樣也是 remap 的部分，現在我們進行 remap 的動作是在 simulation engine 讀入相應模擬設定檔的時候，這樣有三份檔案要改，我們就必須修改三個地方的程式碼，如果日後由於在 NCTUns 上開發新式 protocol，需要 remap 的檔案越來越多，難道我們也一處一處去修改，這樣很沒效率。未來我們希望能夠利用 script 自動為模擬引擎作 remap 的動作，不過由於模擬設定檔格式是由 protocol module 開發者所決定，也許必須統一模擬設定檔對於 IP 與 tunnel ID 的格式，才方便如此自動 remap。

7.3 模擬任務可遷移性

最後一點是在與虛擬機器比較時發想，現在模擬機器能夠很輕易地藉由遷移虛擬機器來達到負載平衡的效果。未來我們希望也能實現一套遷移模擬任務的機制，使得在需要時能夠由負載高的並行模擬伺服器直接遷移模擬任務至負載較低的並行模擬伺服器。



捌、參考文獻

[1] 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, pp. 175-197.

[2] S.Y. Wang, C.L. Chou, C.C. Lin, “The Design and Implementation of the NCTUns Network Simulation Engine,” Simulation Modelling Practice and Theory, 15 (2007) 57-81.

[3] NCTUns Network Simulator and Emulator, available at http://NSL.csie.nctu.edu.tw/nctuns.html.

[4] Chih-Liang Chou, “A Network and Traffic Simulator for Wireless Vehicular Communication Network Research”, PhD thesis, National Chiao Tung University, Hsinchu, Taiwan, 2009.

[5] Liao Kauo-Chiang, “Porting the NCTUns network simulator to Linux and Supporting Emulation”, Master thesis, National Chiao Tung University, Hsinchu, Taiwan, 2004.