國立交通大學

資訊科學與工程研究所

碩 士 論 文

Linux網路通訊協定堆疊之高效率動態的指令嵌入平台之設計與實作

Design and Implementation of an Efficient and Configurable Instrument Platform for Linux Network Protocol Stack

研 究 生：曹敏峰

指導教授：曾建超 教授

中 華 民 國 九 十 七 年 七 月

Linux 網路通訊協定堆疊之高效率動態的指令嵌入平台之設計與實作

Design and Implementation of an Efficient and Configurable Instrument Platform for Linux Network Protocol Stack

研 究 生：曹敏峰　　Student：Min-Feng Tsao

指導教授：曾建超　　Advisor：Chien-Chao Tseng

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

July 2008

Hsinchu, Taiwan, Republic of China

中華民國九十七年七月

Linux 網路通訊協定堆疊之高效率動態的指令嵌入平台之設計與實作

研究生： 曹敏峰　　指導教授： 曾建超 教授

國立交通大學資訊學院資訊科學與工程研究所

摘　要

本論文之目的是在 Linux 作業系統的網路通訊協定堆疊上，設計與實作一套高效率動態之指令嵌入平台，讓使用者發展嵌入式網路通訊裝置之網路核心系統與通訊協定行為的整合測試、分析工具。

近年來，由於各式應用的嵌入式網路通訊裝置需求不斷增加，驅使業界及學術界皆投入大量研發人力參與新產品的開發。就嵌入式網路通訊裝置的網路行為而言，除了基本的網路存取(access)能力外，經常關心的會是傳輸的延遲(delay)、反應時間(response time)以及封包遺漏(packet loss)等相關研究議題，因此如何量測各類的延遲、反應時間以及封包遺漏是許多研發人員必須面對的問題。

然而在目前，只有類似 network sniffer 的通訊封包(packet)擷取工具，能夠協助分析網路通訊系統「外顯的(external)」網路延遲、反應時間與封包遺漏；對於網路通訊裝置系統內部的核心(kernel)協定處理程序(processes)的延遲以及所造成的封包遺漏，現在卻無任何工具能協助研發人員界定其發生的原因。因此，本論文之研究目的即是針對嵌入式之網路通訊裝置，發展一套「可動態嵌入核心函式擷取指令」的平台，以擷取、紀錄與追蹤網路核心函式，利用此平台可進一步結合通訊封包擷取(packet sniffer)工具，發展嵌入式網路通訊裝置之網路核心系統與通訊協定行為的測量、分析與評量系統。藉由此系統，研發人員可輕易分析 Linux 核心網路系統的行為，對網路通訊協定堆疊的每一個重要且具關鍵性之函式做剖析研究並記錄相關訊息，之後再針對所記錄的相關訊息做額外的分析結果報告，藉此找出封包在傳送及接收時所發生的延遲瓶頸以及對整體的網路通訊堆疊做效能分析。

本論文首先提出一個最基本、最直覺的設計概念(Naive Approach)，以了解擷取核心函式的概念、方法以及基本考量，接著再提出本論文所使用的方法來改善 Naive Approach 的缺失。本論文所提出的平台架構，主要分做四大模組: (1) Linux Kernel Patching (2) Instrument Modules (3) Instrument Configuration Interface (4) Log

File Generator and Analyzer。利用這四大模組，本平台可動態嵌入擷取指令，提升整體效能，避免 Naive approach 所會遇到的種種問題。

最後，我們使用實際的通訊程式來測量 Patch 之後的 Linux Kernel 的效能，以及本論文所提出的方法與 Naive approach 對於處理核心網路堆疊速度的比較。實驗結果顯示，本論文所提出的架構確實能夠改善 Naive approach 所遇到的問題，在使用 Linux Kernel Patching 之後對於原始核心的影響非常小。

關鍵詞：嵌入式系統、可攜裝置、網路通訊、Linux、網路通訊協定堆疊、開放原始碼

Design and Implementation of an Efficient and Configurable Instrument Platform for Linux Network Protocol Stack

Student: Min-Feng Tsao Advisor: Dr. Chien-Chao Tseng

Institute of Computer Science and Engineering
College of Computer Science
National Chiao Tung University

Abstract

Due to the continuously increasing demands for various applications of embedded networking and communication devices, the industry and academic both have spent a great amount of research efforts on developing new and better embedded products. In order to develop a high efficient communication system for embedded networking and communication devices, we need to consider not only the basic network accessibility but also the critical performance metrics, such as transmission delays, response times, and packet losses, of embedded devices. Therefore, how to measure various delays, response times and packet losses becomes the major research issue in embedded networking systems.

Many network sniffer tools exist for researchers to intercept and analyze network packets. However, the current network sniffer tools can capture only the "external" networking behaviors of communication protocols but not the effects of network kernel subsystems on communications. In other words, they merely intercept network packets for analysis but do not provide any vehicles for the analysis of kernel effects on communications. Therefore, this thesis plans to design and develop a platform that can help researchers to investigate the integrated effects of both the internal behavior of a kernel subsystem and the external behavior of the protocol packet exchanges.

This thesis focuses on analyzing the behavior of network communication in Linux kernel at first. Then we develop a platform that can record related information about the important or key functions in Linux network protocol

stack. With the recorded information, we can analyze to the networking capabilities and identify the bottlenecks of embedded networking and communication devices. Consequently we can try to reduce the packet transmitting/receiving delays and improve the overall performance of the network protocol stack.

The goal of this thesis aims to develop an efficient and configurable instrument platform for Linux network protocol stack. We first present a naive approach that adopts the basic and intuitional methods in the design of this platform. By presenting the naive approach, we can introduce the concepts, implementation methods and design considerations of kernel patching. We then describe the concepts and design of our configurable instrument platform. The architecture of the platform consists of four basic components: (1) Linux Kernel Patching, (2) Instrument Modules, (3) Instrument Configuration Interface, and (4) Log File Generator and Analyzer. With these four modules, a user can configure instrument profile dynamically, that is at runtime and without recompilation, to trace the kernel behaviors the user is interested at. Furthermore, with the ingenious design of the Instrument Module, our platform can trace kernel functions and record kernel events with just a slightly overhead.

Finally, we use real network protocols in measuring the performance of Linux Kernel Patching mechanisms and compare the effectiveness of our approach with the naive approach. The results show that our configurable instrument platform introduces only negligible overhead, and is much superior to the naive approach.

Keywords: embedded system, portable device, networking and communication, Linux, network protocol stack, open source

誌　謝

首先要感謝曾建超老師這兩年來的指導，以及提供我這麼好的研究環境，讓我可以沒有後顧之憂地完成這篇論文。此外，本論文的原始想法及動機也源自於老師，謝謝老師能夠不厭其煩的提醒我的錯誤及不足之處，讓這篇論文的完整性更加提升。

再來要感謝各位口試委員，包括了曹孝櫟老師、嚴力行老師以及翁永昌老師，在百忙之中抽空來參加我的碩士論文口試，並提供了許多寶貴的意見，特別是曹孝櫟老師對於 Linux 以及嵌入式系統的開發經驗著實讓我有了許多新的體悟。在我研究所的兩年期間，曹老師也一直是我佩服至極的學長(同出身於成大)，讓我見證了對於學術研究該有的熱忱、態度及方法，絕對讓我一輩子都能受用無窮。

感謝實驗室的所有夥伴們，謝謝史老大、RT 學長以及永昇學長在論文及口頭報告上給予我許多建設性的意見；謝謝同屆的夥伴們：宗羲、俊羽和昭哥，無論是在課業上或是撰寫論文的歷程中，都給予了我許多幫助，讓我從沒感覺到是一個人在奮鬥；謝謝我最可愛的學弟妹們，讓我在撰寫論文的碩二期間從來不會感到枯燥乏味，尤其是冠銘、興謙、俊延、媛莉，以及認識不到半年的萱萱，很開心認識了你們。

謝謝大學的朋友們，特別是佩姍與思如，一直在背後默默地給我鼓勵，讓我在求學階段最失意的時候重新燃起動力，也祝福妳們在之後的求學路途中可以一帆風順。最後，最重要的還要感謝我的家人，感謝爸爸媽媽一路上的支持，讓我可以順順利利的走到這裡。

目　錄

圖目錄

表目錄

第一章　緒論

1.1 研究動機

近幾年來隨著電腦科技的進步與網際網路的興起，資訊產品的研發技術不斷地進步與創新，使得多種規格的無線網路(例如 GPRS、WLAN、3G 之 CDMA2000、WiMAX 等)及網路通訊裝置應運而生。此外，為了減少網路通訊裝置的體積、重量與成本，嵌入式系統(Embedded System)技術則扮演著極重要的角色，像是今日隨處可見為因應不同需求的可攜裝置(Portable Device)，例如個人數位助理(Personal Digital Assistant, PDA)、行動電話(Mobile Phone)等，都是利用嵌入式系統來實現這些裝置，另有一類網路通訊裝置為了穩定性及安全性考量，也常以嵌入式系統技術實現，例如行動路由器(mobile router)等，在本論文中則泛稱以上裝置為嵌入式網路通訊裝置(Embedded Networking and Communication Device)。

就一個嵌入式網路通訊裝置的網路行為來說，除了基本的網路存取(access)能力外，我們經常關心的會是傳輸的延遲(delay)、反應時間(response time)以及封包遺漏(packet loss)。傳輸的延遲、反應時間與封包遺漏會受到使用的網路類型、作業系統(operating system)及通訊協定(protocol)等種種因素的影響而有所變化。尤其當嵌入式網路通訊裝置在網路上漫遊(roaming)期間進行與各個無線基台間換手(handoff)時，由應用程式(application program)乃至作業系統等網路通訊協定(protocols)相關程式都得因應換手而有所動作，所以也會造成傳輸延遲與封包遺漏。因此有必要開發一套合適的輔助工具，協助嵌入式網路通訊裝置的開發者正確界定造成延遲與封包遺漏的原因，進而分析問題及改善缺失。

目前，對於網路通訊裝置系統核心的網路通訊協定處理程序(processes)延遲以及核心事件(kernel event)，沒有一套完整的工具能夠協助開發者了解傳輸延遲、反應時間以及封包遺漏的發生原因。雖然目前有類似 network sniffer 的網路通訊封包(packet)擷取工具，卻僅能協助分析外顯的(external)封包類型以及接收時間等，對於核心內部(internal)所引發的網路延遲分析仍較缺乏。因此，我們希望可以發展一套整合核心系統與通訊行為的網路通訊裝置之評比工具，主要整合內容之重點有三：(1)核心內部通訊協定堆疊之分析(2)核心內部網路事件通知機制(3)封包擷取工具。而本論文將著重在第一個部份，即有關核心內部之通訊協定堆

疊分析，而另外兩部分與整合工作將列於未來工作之內容。

1.2 研究目標

本論文的研究目標是發展一個 Linux 網路通訊協定堆疊的分析平台工具，讓嵌入式網路裝置發展者可以利用此平台工具來分析裝置本身的核心網路行為，包括了：

- 封包在核心內的傳輸流程
- 封包在核心內的接收流程
- 封包傳輸時的延遲
- 封包接收時的延遲

此外，本論文將針對效能與功能面上做加強，以期能達到以下兩大重點：

- 就效能面而言，由於現今網路以及硬體的進步，在核心網路內的處理速度已經相當快速，我們在記錄核心資訊的動作很有可能造成核心本身更大的負擔。因此，我們必須針對這部分做效能上的補強，希望能以不影響原始核心通訊協定堆疊之效能為目標。
- 就功能面而言，我們希望能夠提供一個具備動態調整指令嵌入的平台。意即是指當我們在分析 Linux 網路通訊協定堆疊時，可以調整我們要分析的區域部份，像是針對於 TCP 連線的追蹤或是僅觀察網路層(network layer)所有的行為。

1.3 章節簡介

本篇論文的章節編排與內容簡介如下：

第一章：緒論，介紹本篇論文的研究動機及希望達成的目標。

第二章：背景知識介紹，介紹本篇論文所要探討主題以及所應用到的相關知識，主要是針對 Linux 相關的知識，包括作業系統、核心模組以及 proc 檔案系統。

第三章：相關研究，簡介與本篇論文主題相關的文獻及工具，包括了 Linux 核心偵錯工具、封包過濾工具以及本實驗室之前所發展過的 Linux 網路事件擷取相關工具。

第四章：核心網路追蹤工具之設計與架構，說明本論文所開發的系統平台之系統架構與期內各個軟體元件。

第五章：核心網路追蹤工具之實作，說明本論文所開發的系統平台之實作細節，以及針對 Linux 內部所做的修改部分。

第六章：實驗結果分析及貢獻，介紹本系統效能評估的實驗結果分析以及總體貢獻。

第七章：結論與未來工作，總結本篇論文的結果，以及未來可繼續研究的方向。

第二章　背景知識介紹

2.1 Linux 作業系統

Linux 作業系統(嚴格說起來是 Linux 核心，Linux kernel)的創始人是 Linus Torvalds，當初只是在上大學時出於個人愛好而編寫的，第一個發佈在網路上的版本是在 1991 年 9 月，最初 Torvalds 稱這個核心的名稱為"Freax"，意思是自由(free)和奇異(freak)的結合字，後來將核心更名為 Linux，而當時候的程式碼只有約一萬行。[1][3]

Linux 的歷史是和 GNU 計劃緊密聯繫在一起的，所以本段將先從 GNU 開始介紹。GNU 是「GNU's Not Unix」的縮寫，在此計劃提出之時 UNIX 是一種廣泛使用的商業作業系統的名稱，從 1983 年開始的 GNU 計劃致力於開發一個自由並且完整的 Unix-like 作業系統，並包括軟體開發工具、文書編輯器以及各式各樣的應用程式。之後在 1985 年又創立了自由軟體基金會(Free Software Foundation)來為 GNU 計劃提供技術、法律以及財政支援。此外，為保證 GNU 軟體可以自由地「使用、複製、修改和發佈」，所有 GNU 軟體都包含一份在禁止其他人添加任何限制的情況下，授權所有權利給任何人的協議條款，即「GNU 通用公共許可證(GNU General Public License，GPL)」。[2][3]

到了 1991 年 Linux kernel 發佈的時候，GNU 已經幾乎完成了除了系統核心之外的各種必備軟體，唯獨「GNU’ s kernel」並沒有完成，故在當時選擇了 Linus Torvalds 所發佈的 Linux kernel 作為搭配。在 Linus Torvalds 和其他開發人員的努力下，GNU 組件可以運行於 Linux 核心之上，整個 Linux 核心是基於 GNU 通用公共許可，也就是 GPL(GNU General Public License，GNU 通用公共許可證)的，但是 Linux 核心並不是 GNU 計劃的一部分。在 1992 年 Linux 與其他 GNU 軟體結合，完全自由的作業系統正式誕生。該作業系統往往被稱為「GNU/Linux」或簡稱 Linux。[2] [4]

而現今 Linux 作業系統通常指的是「Linux 套件(Linux distribution)」，它可能是由一個組織，公司或者個人發行的。一個 Linux 套件除了包括 Linux 核心之外，還包括一套安裝工具、各種 GNU 軟體、其他的一些自由軟體，在一些特定的 Linux 套件中也有一些專有軟體。在目前的階段中，使用者較多的套件諸如：Ubuntu[5]、Fedora[6]等。

Linux 目前為最受歡迎的伺服器作業系統之一，此外，Linux 系統也開始慢慢搶佔桌面電腦的作業系統市場。近幾年來，隨著網路的發展與硬體的進步，嵌入式系統逐漸受到市場的重視，也有越來越多嵌入式系統的產品被發展出來，像是手機、PDA、遊戲機以及最近走紅的 Eee PC 等，由於 Linux 作業系統低成本加上許多程式皆為開放原始碼(open source)容易取得的特性，可以想見 Linux 在嵌入式電腦市場上擁有絕對的優勢將使得 Linux 系統倍受廠商與用戶的青睞。

本論文也是針對目前嵌入式的網路設備日益增多的趨勢，加上 Linux 作業系統開放原始碼的特性，故選擇以 Linux 作業系統作為主要的研究分析平台，希望在未來能夠在以 Linux 作為開發平台的嵌入式網路設備上有些貢獻。此外，本論文主要的研究主題是指 Linux kernel 中網路通訊協定堆疊的部分，GNU 程式以及 Linux 作業系統本身的其他部分除了用來協助本論文研究的工具之外，有一部分將作為本論文的系統架構的主要元件。

2.2 Linux 核心模組

Linux 核心模組(Linux kernel module)嚴格說起來應稱作 Linux 動態載入的核心模組(Linux Loadable Kernel Module, LKM)，有時候也簡稱 module。實際上模組可能包括內建於核心內部的模組或是另外撰寫以供未來以掛載方式載入的模組，而在此章節中我們所探討的 Linux 核心模組所指的是後者，以動態載入方式的核心模組。

有許多人會說「核心模組位在核心之外」、「核心模組與核心(kernel)溝通」，這些都是不正確的[7]。實際上，Linux 核心模組是核心的一部分，當核心開始工作時，Linux 核心模組在掛載之後也屬於核心的一部分程式碼，在某些作業系統中，甚至有人也稱 Linux 核心模組是核心的延伸體(kernel’ s extension)。此外，核心模組與核心(kernel)溝通也是不完全正確的，正確點來說應該是與最基本的核心(base kernel)溝通，而此最基本的核心指的是核心一開機就被編入映像檔的最基本部分(The part of the kernel that is bound into the image you boot)[7]，並不包括那些後來被載入的 Linux 核心模組(LKMs)。

Linux 核心模組當初被發展出來的動機是在於若所有的裝置驅動程式(Device driver)都包在 Linux 核心中勢必會讓核心大小變得很大，若要編製一個專屬於某特定電腦的核心也是相當不便利的，此後，便有人希望可以將核心只納入最基本的功能，選擇性的功能或支援轉以 Linux 核心模組來完成。

Linux 核心模組的主要特色便是提供了一種非常有效的方法，在不需要重新編譯核心之下便可延伸基本 Linux 核心的功能。這在我們想要加入某個新功能到核心中的時候特別有用，既可以不加大原本核心的大小，也可以在不用重新編譯及重新開機下便載入此新功能[8][9][10]。此外，核心模組可以在需要某功能時才掛載此模組，而不再需要此功能之後也可以卸載此模組以還原到原始狀態。

就目前而言，Linux 核心模組多半用於以下六種情形[7]：

- 裝置驅動程式(device driver)：裝置驅動程式是為了某些特定的硬體所寫的，首要目的便是要讓 Linux 核心能與這些硬體設備做溝通，要使用某個裝置設備，在 Linux 核心中都必須要有一份對應的裝置驅動程式。而裝置驅動程式其實並不需要對硬體如何運作了解太多，因為所有使用 Linux 核心模組來操作硬體的驅動程式都有其固定的撰寫方法，詳情可參照[29]。
- 檔案系統驅動程式(Filesystem driver)：一個檔案系統驅動程式主要的目的是用來解讀某一種特定的檔案系統格式，解讀後便是我們所熟悉的目錄以及檔案結構。檔案系統像是一般 Linux 作業系統常用的 ext2 filesystem，或是其他諸如 MS-DOS 或是 NFS 等，都必須要擁有一個檔案系統驅動程式才能夠解讀。
- 系統呼叫(System call)：系統呼叫的目的是為了讓 user space 的程式能夠取得 kernel space 中的服務(service)，像是讀取檔案、開啟一個新的行程(process)或者重新啟動及關機等。而一般標準的系統呼叫都是整合進基本的核心(base kernel)之中，不是以 Linux 核心模組的形式編進 Linux 核心之中，然而，Linux 核心模組也提供了自行設計系統呼叫的方式，讓使用者可以發明某些特定的系統呼叫或是覆寫目前系統提供的系統呼叫。
- 網路驅動程式(Network drivers)：網路驅動程式是用來實作一個網路通訊協定，用在 Linux 通訊協定堆疊之某層中來傳送及接收資料串流(data stream)。舉例來說，如果想使用 IPX 通訊協定來處理網路串流，必須擁有一個 IPX 的驅動程式。
- tty 管制線(tty line discipline)：tty(古老 teletypewriter 的縮寫)在現今是指所有類似串列埠(serial port)的裝置，舉凡串列埠(serial port)、USB-串列埠轉換器(USB-to-serial-port converter)以及某些數據機(modem)

都屬此類。而終端機裝置(terminal device)可以透過此 tty 裝置來進行資料傳輸，然而，在終端機裝置要傳輸時，tty 裝置也必須要有裝置驅動程式(device driver)才能夠使用。tty 裝置驅動程式主要被切成三個部分，示意圖如 Figure 2-1：tty core 主要是負責控制資料流向以及資料的格式，這樣可以讓 tty driver 專心地處理與硬體溝通之資料傳輸，而不需要去擔心如何與 user space 溝通。對一般的 tty 裝置而言，其實是不需要經過 tty line discipline driver 的，但對於某些特定通訊協定是需要額外對資料格式做轉換的，像是 PPP(Point-to-Point Protocol)或是藍芽(Bluetooth)，便需要 tty line discipline 來協助轉換資料格式，這部分便可以以 Linux 核心模組的方式來撰寫，詳細介紹可參考[29]。

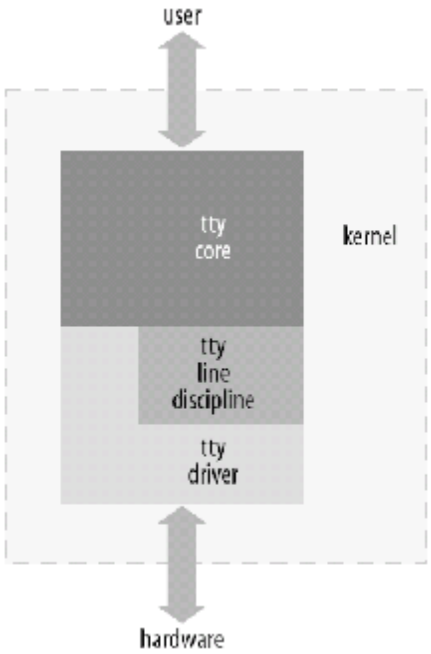

Figure 2-1 TTY Core 總覽[29]

- 執行檔解讀器(executable interpreter)：執行檔解讀器是用來載入並執行一個執行檔，在現在的 Linux 作業系統中最常見的是 ELF(executable and linkable format)執行檔，但其實 Linux 作業系統的設計上可以執行多種執行檔，只是每一種執行檔都必須有一個執行檔解讀器才能執行。Linux 核心模組也是撰寫執行檔解讀器的一種方法。

至於要如何撰寫一個 Linux 核心模組呢？一個 Linux 要完成編譯除了模組本身的程式碼之外，還需要一個用來編譯的 Makefile 檔案，以下將分別介紹這兩個部分：

```
#include <linux/module.h>
#include <linux/kernel.h>

int init_module(void)
{
        printk(KERN_INFO "The module was loaded.\n");
        return 0;
}

void cleanup_module(void)
{
        printk(KERN_INFO "The module was removed.\n");
}
```

Figure 2-2 Linux 核心模組的簡單範例

一個基本的 Linux 核心模組首先必須包含 module.h 這個標頭檔，而在 Figure 2-2 中可以看見範例還有包含另一個標頭檔 kernel.h，此檔案是因為其後的程式碼使用到 printk 的核心輸出函式。此外，所有的 Linux 核心模組都有一個開始的初始化函式如 init_module()，負責處理在掛載此核心模組時的初始化動作；也有一個對應的結束函式如 cleanup_module()，負責處理在此核心模組相關的記憶體釋放動作。然而，自從 Linux 2.4 版之後，使用者可以自行對這兩個函式重新命名，雖然目前仍有許多開發者沿用此命名，詳細修改的方式可見[8]。

```
obj-m := myModule.o

all:
        make -C /lib/modules/$(shell uname -r)/build \
        M=$(PWD) modules
clean:
        make -C /lib/modules/$(shell uname -r)/build \
        M=$(PWD) clean
```

Figure 2-3 基本核心模組之 Makefile

Figure 2-3 則顯示了一個針對於 Linux 核心模組所撰寫的編譯檔案 Makefile，而實際上只有第一行是必要的，即產生所要的模組檔案名稱；而" all" 以及" clean" 的選項是為編譯上的方便性才加入的。在使用 Makefile 編譯完畢之後會產生模組檔案 myModule.ko (Linux 2.6 之後使用.ko 副檔名取代 Linux 2.4 所使用的.o 副檔名)，此後便可以利用 modutils (module utilities) 的相關工具來執行掛載(insmod)、卸載(rmmod)以及察看(lsmod) Linux 核心模組。

Figure 2-4 顯示了使用 Module(Linux 核心模組)與 Core Kernel(核心)之間的

關聯性，Module 包括初始函式 module_init()以及離開函式 module_exit()，其中內部還包括許多另外撰寫的功能函式(Module functions)。當使用掛載工具(insmod)之後，Module 便會執行 module_init()函式進行初始化動作，之後會把指向 Module functions 的一個指標當作參數傳入 Core Kernel 的 register_capability()函數中，此函數會把指標放入一個 capabilities 陣列的資料結構中，接著系統便會定義一些通訊協定的方法讓使用者可以透過系統呼叫(System Calls)來操作這個 capabilities 資料結構。最後透過卸載工具(rmmod)來執行 Core Kernel 中的 unregister_capbility()函式並釋放 capabilities 資料結構。

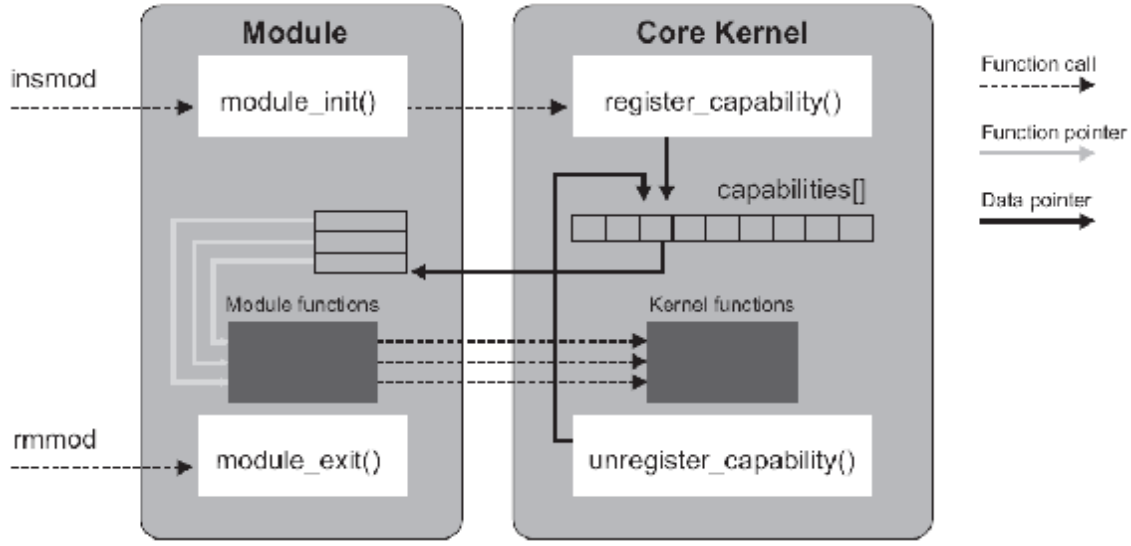

Figure 2-4 使用 Linux 核心模組連結到核心[9]

其他更多有關 Linux 核心模組的細節，例如使用參數傳遞給 Linux 核心模組、切割多個檔案合併成一個 Linux 核心模組等，以及 Linux 核心模組與核心間的詳細溝通機制，可以參考[8][9]。

2.3 procfs (/proc filesystem)

/proc file system 是一個虛擬檔案系統(virtual file system)，位於目前 Linux 檔案系統之下的/proc/*，當初/proc 檔案系統設計的目的是要更容易地的觀看有關行程(processes)的資訊，所以因此有了/proc 這個名稱，而現今則是常常用來顯示一些系統資訊，例如 CPU、記憶體使用狀態的資訊或是系統模組等，如 Figure 2-5 查看/proc/meminfo 這個虛擬檔案來了解記憶體的使用資訊。

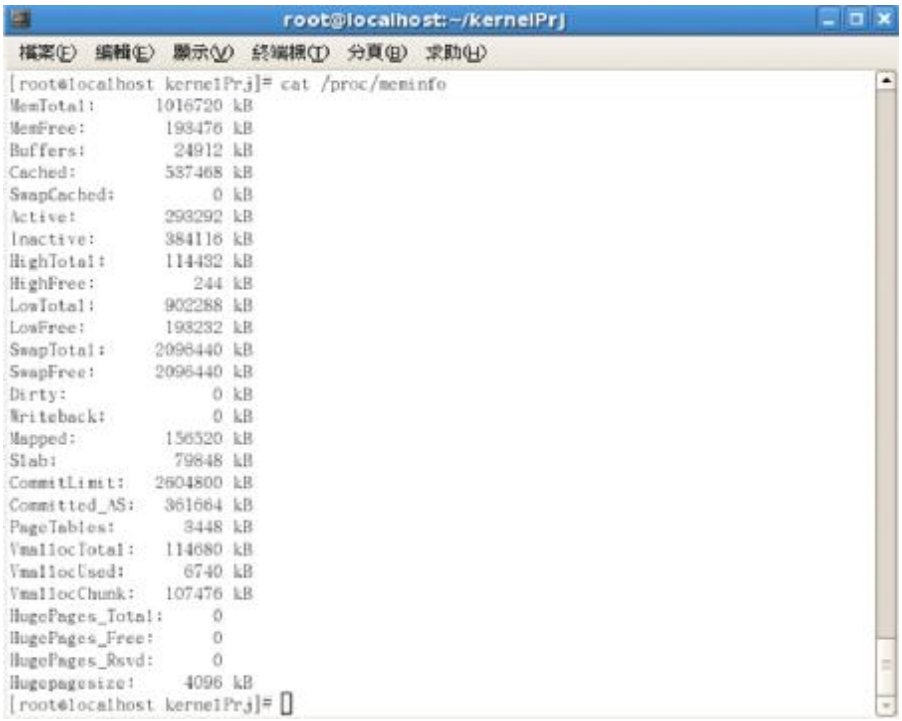

Figure 2-5 使用 procfs 顯示記憶體資訊

/proc 檔案系統實際上是直接利用記憶體的空間來儲存這些資訊，所以當系統關機的時候，這些位於/proc 檔案系統的訊息即會被清除，一旦重新啟動系統的時候，整個/proc 檔案系統會被重新建立起來。此外，由於/proc 檔案系統是顯示記憶體的一些資訊，所以實際上是不會占用硬碟空間的，也因為執行時期都是使用記憶體在操作，避免透過系統 I/O 的動作，使整體的系統負擔相對減少了許多，唯有在使用者在讀取或寫入此/proc 的檔案系統時，才會使用到系統 I/O 來處理，這部分在之後會詳加介紹。

/proc 檔案系統的使用方法與撰寫方式其實與裝置驅動程式很類似(裝置驅動程式的詳細內容可以參考[29])，一開始需要定義一個完整的資料結構來存放所有/proc 檔案所需要的處理函式(handler function)的指標，例如當我們讀取或寫入/proc 檔案時所要處理的函式。之後，如同 Linux 核心模組的寫法(可參考 2.2Linux 核心模組)，在 init_module()的部分註冊此結構，於 cleanup_module()的部分反註冊，以下將以程式範例說明：

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h> /* Necessary when using the procfs */

struct proc_dir_entry *Proc_File;

int init_module()
{
        Proc_File = create_proc_entry("helloworld", 0, NULL);
        if (Proc_File == NULL) {
                remove_proc_entry("helloworld", 0);
                return .ENOMEM;
        }

        Proc_File->read_proc = procfile_read;

        return 0;
}

void cleanup_module()
{
        remove_proc_entry("helloworld", 0);
}
```

Figure 2-6 proc 檔案系統程式範例-part1

Figure 2-6 顯示了一個標準的 Linux 核心模組寫法，包括了初始化函數 init_module()、結束函數 cleanup_module()，以及所需用到的 module.h 和 kernel.h 標頭檔案，因為/proc 檔案系統的使用方式多數是以 Linux 核心模組的方式來撰寫的，所以仍須遵照此方法。此外，在開始/proc 檔案系統之前必須先引入 proc_fs.h 的標頭檔，在 init_module()初始化時執行 create_proc_entry()這個函式，便會產生系統的虛擬檔案/proc/helloworld，而此函數的回傳值便是描述此檔案的資料結構 Proc_File，在之後的程式碼中指定了此結構的讀取函數是 procfile_read，此部分的程式碼可見 Figure 2-7，而實際上並不只讀取函數可以設定，另外還包括了寫入函數以及此檔案的一些設定資訊，但在此部分我們只針對讀取函數作為範例，詳細撰寫方法可參考[8][9]。最後，在 cleanup_module()結束函數時執行 remove_proc_entry()函式來反註冊這個/proc 檔案。

```
int procfile_read(char *buffer, char **buffer_location,
                  off_t offset, int buffer_length, int *eof, void *data)
{
        int ret;

        if (offset > 0) {
        /* we have finished to read, return 0 */
                ret = 0;
        }
        else {
        /* fill the buffer, return the buffer size */
                ret = sprintf(buffer, "HelloWorld!\n");
        }
        return ret;
}
```

Figure 2-7 proc 檔案系統程式範例-part2

Figure 2-7 則是承接上個部份的程式碼，顯示了讀取函數 procfile_read 的部分，當我們讀取此虛擬函數的時候(如使用 cat 工具)，便會執行此函數。此函數共有六個參數，而在此範例中我們只針對第一個參數 buffer 做設定：buffer 是核心系統預先幫我們配置的一個記憶體區塊，用來存放此/proc 檔案所要輸出的訊息，所以可以直接將要輸出的訊息”HelloWorld!\n”寫進這個 buffer。此外，若要使用自行配置的記憶體區塊則可以利用第二個參數 buffer_location 指標，指向我們自行配置的 buffer 位置，而第三個參數 offset 則代表了目前這個檔案所讀取到的位置。另外一個重點是：此函數的回傳值代表了還有多少資料數量要輸出，若回傳值為零時，則代表此檔案已經輸出完畢。

之後編譯完成並掛載此模組的時候便會產生/proc/helloworld 檔案，在讀取此檔案時，如”cat /proc/helloworld”，便會輸出”HelloWorld”的訊息，而在卸載此模組的時候，便會刪除/proc/helloworld 檔案，以上便是/proc 檔案系統的運作流程。

第三章 相關研究

本章將會介紹本論文的相關研究，包括了(1)Linux 核心偵錯工具(2)封包過濾工具以及(3) Linux 核心網路事件通知機制。在目前的研究中與本論文的研究目標相吻合的其實不多，最具相關性的代表是[11]，該論文針對於 Linux 2.0 的版本之 TCP/IP 網路堆疊進行效能評估。而本章中之每小節都是針對於 Linux 底下與本工具平台類似的相關工具做探討。

3.1 Linux 核心偵錯工具

一般的偵錯工具(Debugger)主要分作兩個類型：

- Kernel-level Debugging
- User-level Debugging

Kernel-level Debugging 主要的針錯對象是所有在 kernel space 所執行的程式碼，包括正在使用的 Linux 核心以及所有 Linux 核心模組；而 User-level Debugging 的主要對象是針對在 user space 執行的應用程式，可以再細分為單一工作/執行緒(task/thread)程式、多工作(multi-tasking)程式以及多執行緒(multi-thread)程式。在這章節中我們主要是針對 Kernel-level Debugging 做探討，因為本論文的主要分析對象是 Linux 核心程式碼，所以 Kernel-level Debugging 這部分的相關研究也較有參考價值。

對於 kernel space 中所使用的偵錯工具都有其特定之目的：gdb(GNU Debugger)[12]主要是適用於原始碼層面(source-level)的偵錯，支援執行緒、遠端除錯以及硬體架構模擬等，也包括了互動式的偵錯功能，如中斷點、watch、逐步偵錯等功能，而 kgdb[13]則是對於 gdb 提供核心層面的擴充；kdb(Built-in Kernel Debugger)是內建於 Linux 核心中的偵錯工具，主要是針對組合語言層次(assembly-level)上的偵錯；kerneloops[15]則是針對 Linux 核心發生例外(exception)時能夠產生錯誤訊息，將經由 klogd 輸出到 kernel ring buffer；kprobe[16]可以針對 Linux 原始碼設置中斷點，並將中斷點所在的函式安插一小部分的程式碼來達到輸出相關變數及訊息之功能。

對於目前的 Linux 核心偵錯器而言，其實已經提供了許多功能強大的除錯功能，但其主要功能還是針對核心發展者在發展核心以及相關模組等時期所使用，

且此些偵錯工具背後所使用的機制與所使用的記憶體資源以及系統輸出入(system I/O)並不是十分容易掌握。然而，此些偵錯工具所用來觀察 Linux 核心網路的行為上仍具有非常大的幫助，但倘若要利用這些工具來更進一步地開發本論文的系統平台恐怕仍有許多效能上的因素需要考量。

3.2 封包過濾工具

封包過濾器(packet sniffer)又稱網路過濾器(network sniffer)、網路分析器(network analyzer)、通訊協定分析器(protocol analyer)等。通常是一套電腦軟體或硬體在某個特定網路下進行封包的擷取並記錄各個封包的訊息，封包過濾器在擷取了每個封包之後將會針對各封包進行解析，並將封包內容依據對應的 RFC (Request For Comments)有條理地顯示出來。這部分所針對的封包過濾工具是指電腦軟體的部分，將介紹發展一個網路封包監看程式的主要內容。

撰寫一個封包過濾工具，主要有四個重點：

第一，開啟一個可接收 raw packet 的 socket：

一般我們開始 socket 接收 TCP 或 UDP 的 packet 時，我們收到的內容就直接是資料內容，核心已經幫我們把 ethernet 標頭(header)和 IP 或 ARP 標頭都拿掉了。但是在寫網路封包過濾程式時，我們需要標頭，因此我們在開啟 socket 的時候，就可以針對我們要收集的封包總類做過濾，要求接收到的資料裡要包含完整的封包標頭檔。

第二，設定 Promiscuous 模式接收所有封包：

網路卡只會接收到 Ethernet 標頭的目標 MAC 位置(target MAC)是自己的 MAC 位置時，或接收到廣播封包，即目標 MAC 位置為 ff:ff:ff:ff:ff 的封包，或是 multicast 的封包(也就是目標 IP 是 224 開頭的封包)，這種情況稱做 nonPromiscuous。當我們要監看網路封包時，希望收到的不只是廣播、multicast 或是自己的封包時，就需要把網路卡設定成 Promiscuous，如此一來我們就可以接收到實體網路上的所有封包了。

第三，過濾器(filter)的設定：

當我們把網路卡設成 Promiscuous，而且看到了所有封包標頭時，我們就可以依照我們自己的意思去過濾出我們想要的封包。可是主要的問題在於網路上的封包太多，我們全都要一個個的去打開封包標頭，看是否為自己要觀察的封包，

但由於所要花的時間太長，導致後面進來的封包都塞在緩衝器(buffer)裡，等待一個個的封包檢查被檢查完畢。所以我們需要在硬體層(PHY layer)和聯結層(MAC layer)中建立一個過濾器，先過濾出我們要的封包，再交由程式處理，這就是過濾器的功用。而過濾器是由 BPF code 所寫的，詳細資料可參考[17]。

第四，I/O multiplexing (blocking 與 noblocking)

當我們在等待接收網路封包的時候，如果沒有資料進來，程式就會等著並停住，這種情況便稱作 blocking。但有時候我們不想要無止限的讓程式等待，卻又必須跳出這個部份時，除了使用訊號(signal)機制可能可以解決之外，另一個解決方式就是把資料流做 unblocking。但 unblocking 卻不能切確解決我們的需求，而 I/O multiplexing 卻能以較適當的方式來解決這個問題。詳細資料可以參考[18]在 I/O multiplexing 的章節。

目前封包過濾工具已經發展地相當成熟，主要的分析對象以傳統的 Ethernet 有線網路封包到 802.11 的無線網路封包皆可，而通訊協定無論是有線網路及無線網路的封包也幾乎都能夠被解析出來。知名的封包過濾工具如 Wireshark[19]、AiroPeek NX[20]及 Sniffer Portable[21]等。

3.3 Linux 核心網路事件通知機制

此部分將介紹本實驗室之前的一些相關研究，包括了周大鈞之「Linux 平台上驅動程式層網路事件通知機制之設計與實作」[22]與許凱程之「支援具網路感知應用程式的中介軟體之設計與實作」[23]，都是針對於 Linux 平台下發展有關核心空間(kernel space)與使用者空間(user space)溝通的一些機制，以下將分別說明。

周大鈞論文[22]中，使用類似傳統 UNIX 作業系統的信號(signal)機制將網路介面相關的狀態改變通知使用者空間的程式，此外也針對排程演算法做了一些修改，使得使用者空間的程式不需要一直發送系統呼叫(system calls)來獲得這些狀態改變的資訊，加快了對於網路介面處理狀態改變的處理，進而增進了整體的效能。

當網路介面驅動程式發生相關訊息之後，該系統的目標是要從核心空間切換至使用者空間來執行。以信號傳遞的機制下，會在網路驅動程式層產生相關的訊息之後發出信號，通知使用者空間的程式來處理，接著便會執行一個對應於此信號的回叫函式(callback function)。所以當行程從核心空間要跳回使用者空間之前，該系統會先去檢查此行程是否有註冊的事件產生，當有註冊的事件發生時才

會跳至使用者空間去執行對應處理函式，最後再利用系統呼叫返回核心空間。

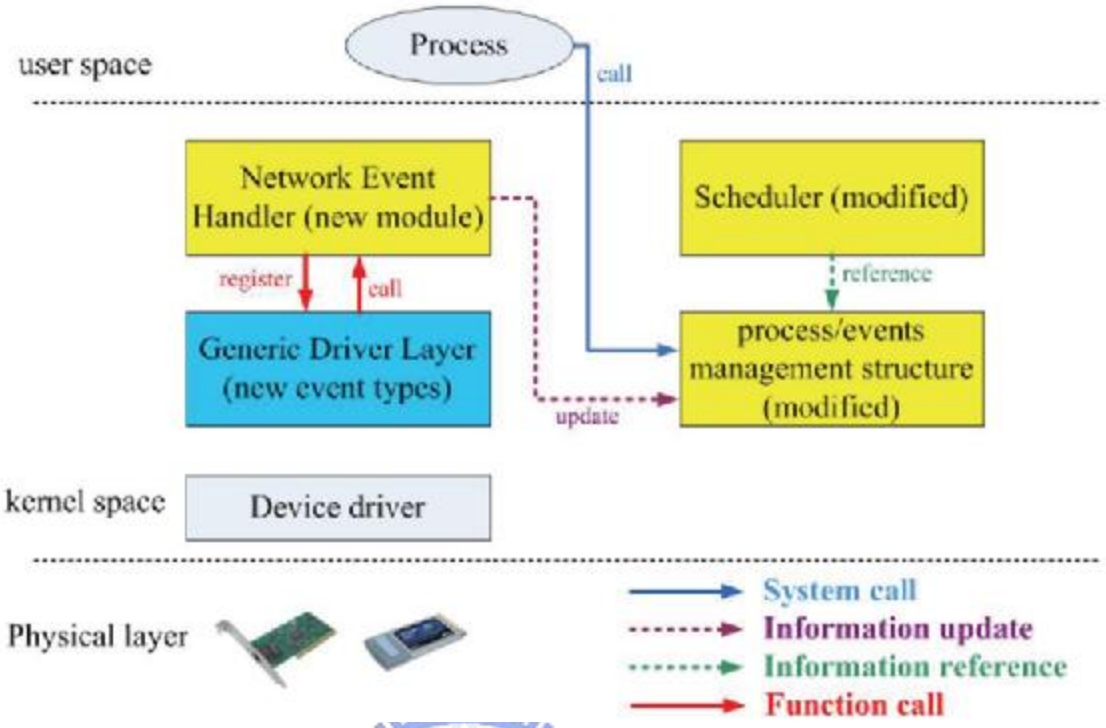

Figure 3-1 驅動程式層網路事件通知機制的系統元件架構與關係圖[22]

Figure 3-1 為整體系統的運作流程。當網路事件發生時，通用驅動程式(Generic Driver Layer)會呼叫 Network Event Handler，Network Event Handler 就會去更改核心用來處理程序及事件的資料結構，標記為某個網路事件已發生，之後當排程器排到該程序時會先檢查是否有事件發生，有的話就會去執行程序所註冊的回叫函式。

許凱程論文[23]中，主要是要設計一套軟體發展平台架構，讓程式開發人員可以利用此平台更方便且快速地開發具網路感知(Network-Aware)的應用程式。

由於現今可搭載多個網路模組的手持網路裝置日漸普遍，如一般常見的模組(WLAN、GPRS)手持裝置，因此目前的環境已經可以讓使用者在各個地點使用不同的方式連上網路。為了讓使用者可以方便地在不同網路間切換，必須設計一個行動管理程式來做智慧型的換手決定(handoff decision)。而以往的行動管理程式必須針對不同的系統撰寫額外的程式碼來處理底層的一些命令，如路由表(routing table)的設定、網路協定堆疊(network protocol stack)的設定以及硬體的控制等，這對程式設計人員都是不小的負擔。

該論文的主要目標便是把這些底層的控制指令以及網路相關的事件抽離出來，而用一個統一的應用程式介面將系統底層的部分隱藏起來，此部分稱作中介

軟體(middleware)平台，對於程式開發者而言就不需要花太多時間去處理各種系統底層的相關程式碼，而可以專注在本身邏輯的開發。對於程式開發者不熟悉系統底層運作流程的人員，將可以大大的避免程式出錯的機會，此部分的相關討論可以參考[24]。

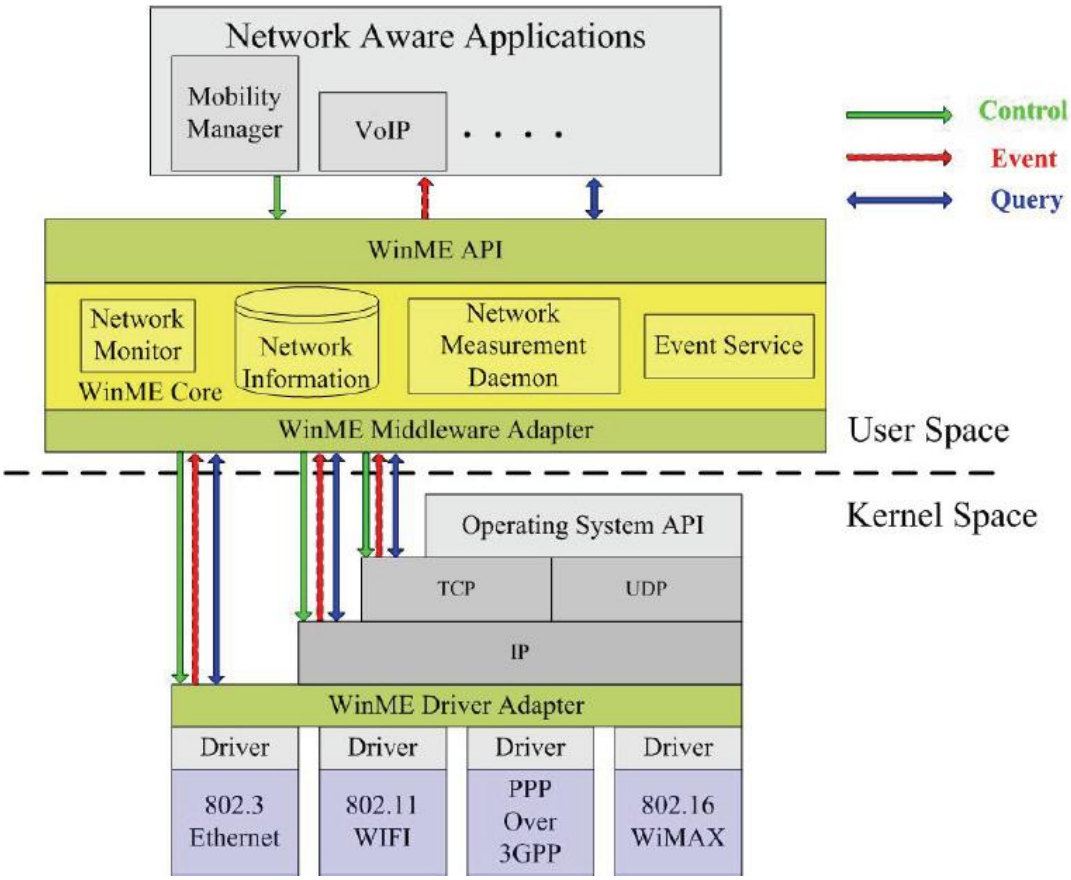

Figure 3-2 中介軟體系統架構圖[23]

Figure 3-2 為該論文之系統架構圖，在核心空間(Kernel Space)中包括了最底層的各種網路介面(Ethernet、WIFI、PPP Over 3GPP、WiMAX)以及對應的 Driver，而此系統將用一個整合所有 Driver 的介面(WinME Driver Adapter)來接收並控制底層各界面的相關訊息，另外並包括了整個網路通訊協定堆疊以及作業系統本身提供的 API(Application Program Interface)；在使用者空間(User Space)則包括了中介軟體的核心(WinCE Core)，將與核心空間中的網路通訊協定堆疊與 WinCE Driver Adapter 溝通，包括了命令(control)、事件通知(event)以及查詢(query)三種方式，而最上層的網路感知應用程式(Network Aware Applications)將可以利用這個中介軟體來與底層的網路相關元件進行溝通，而本身則不需要了解細節。

在 Linux 系統下已經有一個機制可以將核心層的訊息傳送到使用者空間的應

用程式中，這個機制就是 Netlink Socket[25]，而建構在 Netlink 上的 rtnetlink 更可以將核心內部的網路事件以訊息的方式傳送給使用者空間的應用程式。該論文便是採用 rtnetlink 來將核心層的網路事件傳送到事件服務元件，但也由於 rtnetlink 並沒有完全支援所有的網路事件型態，所以該作者針對此部分修改了核心原始碼來補強所有想要觀察的網路事件。

第四章 核心網路追蹤工具之設計與架構

本章將介紹論文中 Linux 核心網路通訊協定堆疊追蹤工具(Linux kernel network protocol stack tracing tool)的設計架構。首先會先對此工具的設計概念及設計目標做介紹，此後再介紹完整體的系統架構之後，將會針對架構中的每一個元件(component)做更詳細的描述。

4.1 設計概念與目標

Linux 核心網路通訊協定堆疊追蹤工具是設計用來分析核心網路的主要行為以及獲得一些通訊協定與核心(kernel)互動的相關重要資訊，諸如時間標記(time stamp)、封包佇列長度等，之後再利用這些資訊來針對核心網路在傳輸及接收過程中的效能做更進一步的分析。

與一般的封包過濾器(Packet sniffer, 例如 Wireshark 軟體[19])不同的是：封包過濾器僅能協助分析通訊協定行為，以及網路通訊系統整體外顯(external)的網路延遲(network delay)、反應時間(response time)與封包遺漏(packet loss)，對於網路通訊裝置系統內部的核心處理程序(kernel processes)的延遲以及所造成的封包遺漏，卻無任何工具能協助界定發生的原因。因此，本論文著眼於能夠發展一套針對於 Linux 網路核心的分析工具，希望能夠彌補目前網路通訊分析工具的不足。

選擇 Linux 作為發展平台的主要原因是因為 Linux 作業系統本身在網路功能已經發展地相當成熟，也支援大多數的網路通訊協定；此外，由於 Linux 作業系統的開放原始碼(open source)，所以可以很容易地觀察到核心內部的行為，諸如核心內部的資料儲存結構、網路通訊系統與核心的互動如中斷(interrupt)機制與裝置輸入輸出(device I/O)行為等，都可以由原始碼直接剖析出來。在現今以 Linux 作為發展平台的工具也不在少數，且因發展的系統或工具都必須遵照 GPL[26]，所以彼此間的經驗與技術都可以在網路上互相交流與學習，因此，本論文所發展的工具與原始碼也將會遵循開放原始碼的形式。

本論文的主要目標將會分析 Linux 核心網路系統的行為，對網路通訊協定堆疊的每一個重要且具關鍵性的函式做剖析並記錄相關訊息，之後再針對所記錄的相關訊息做額外的分析結果報告，藉此找出封包在傳送及接收時所發生的延遲瓶頸以及對整體的網路通訊堆疊做效能分析。以下將先提出一個最基本的設計概

念，了解此方法的缺點與不足之後，再接著提出本論文所使用的方法來改善缺失。

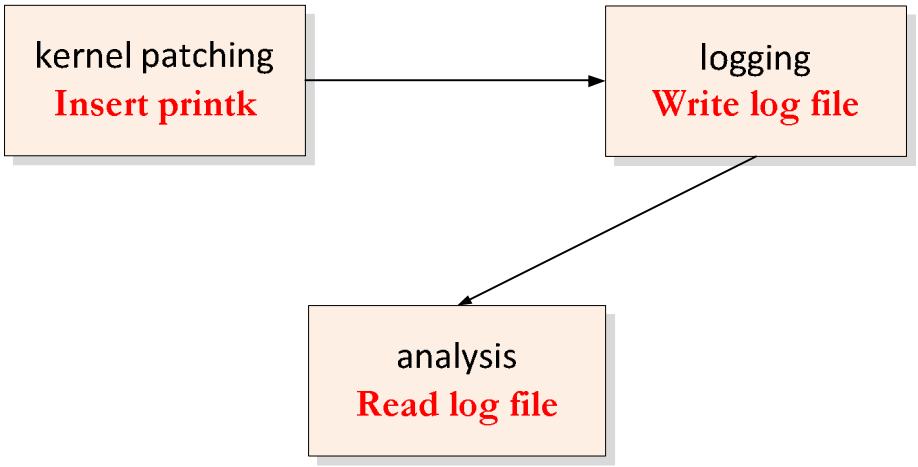

Figure 4-1 Naive approach

Figure 4-1 提出了一個很簡單的基礎概念，在此稱作 Naive approach，共有三個部分：

- 核心補釘(Kernel patching)
- 記錄(Logging)
- 分析(Analysis)

第一個部分是核心補釘(Kernel patching)：使用 Linux 作業系統本身提供的核心函式 printk，可以在 kernel space 中將系統上的變數值印出來。printk 還有所謂的 log level 可以幫訊息做分級的動作，如警告訊息(KERN_WARNING)、嚴重訊息(KERN_ALERT)等，可以說是這些訊息的優先順序(priority) [8][27]，定義在 Linux kernel 原始碼中的/include/linux/kernel.h 檔案。printk 的使用方法類似 C 語言的 printf，配合指定好的 log level 之後，可以將想要印出的訊息輸出到特定的 log 檔案。下面舉一個範例來說明如何使用 printk 來追蹤核心的網路相關函式：

```
00204: static inline int ip_finish_output(struct sk_buff *skb)
00205: {
00206:     printk(KERN_INFO, "function name: ip_finish_output");
00207:
00208: #if defined(CONFIG_NETFILTER) && defined(CONFIG_XFRM)
00209:     /* Policy lookup after SNAT yielded a new policy */
00210:     if (skb->dst->xfrm != NULL) {
00211:         IPCB(skb)->flags |= IPSKB_REROUTED;
00212:         return dst_output(skb);
00213:     }
00214: #endif
00215:     if (skb->len > dst_mtu(skb->dst) &&
00216:         !(skb_shinfo(skb)->ufo_size || skb_shinfo(skb)->tso_size))
00217:         return ip_fragment(skb, ip_finish_output2);
00218:     else
00219:         return ip_finish_output2(skb);
00220: }
```

使用printk輸出此核心網路函式的相關訊息

Figure 4-2 使用 printk 追蹤核心網路函式

Figure 4-2 展示了在網路層(Network layer)中的一個重要函式 ip_finish_output，圖中加了一行 printk 指令，此刻若執行到這個函式的時候便會將此 printk 的訊息" function name: ip_finish_output"，以 KERN_INFO 的 log 層級將訊息輸出到特定檔案，以完成最基礎的核心網路函式追蹤功能。

第二個部份是記錄(Logging)：配合第一個部分選擇 printk 的 log level(在本篇論文中選擇 DEFAULT_MESSAGE_LOGLEVEL)，會將所有 printk 的訊息全部輸出到/var/log/message 中，然而，由於原始作業系統也會將系統除錯及重要訊息的 kernel log 輸出到這個檔案，所以在發展此工具的時候，還必須針對 log 檔案另外執行過濾(filtering)的動作。在第一個部份的核心補釘完畢之後，執行指定的網路功能或通訊協定，便會將有 patch 到的核心函式全部記錄到 log 檔案中，完成第二部份的工作。

第三個部分是分析(Analysis)：最後一個步驟便是針對 log 檔案來做分析，而分析的資訊必須由一開始核心補釘的部分就決定好要輸出的訊息內容為何。本論文主要是針對核心在處理網路封包時傳送及接收的過程做分析，所以所經過核心網路通訊協定堆疊的重要函式都會被記錄下來；此外，為了瞭解網路在傳送及接收經過網路各層所花費的時間，所以在每個函式被執行到的時間點也都會被精準地記錄。在此第三部份的分析工作，第一個重點是要讓使用者了解每個封包在不同的通訊協定之下流經核心網路的過程，無論是在傳送或是接收端；第二個重點是分析網路封包在傳送及接收過程所經過各個函式的時間點，藉以找出網路內部的系統延遲。

然而，使用 Naive approach 這樣的方式仍有許多缺點待改進：

- 第一部分的核心補釘，在每次修改完畢之後都必須要以下動作：(1)重新編譯核心、(2)安裝新核心於 bootloader、(3)最後再重新開機啟動新的核心。假設我們想要修改某個核心函式的輸出資訊，抑或是想要新增對某個核心函式的監看，我們就必須重複這個循環動作，才能使用修改完畢的核心來執行我們想要的分析動作。
- 每當我們執行完核心補釘之後，便無法調整想要監看的核心函式。假設今天我們想要只想要監看網路層(Network layer)的重要函式，來分析封包在傳送及接收的過程，而先前的補釘卻對整個核心網路堆疊都執行了 patching 的動作，這時候必須重新調整核心補釘的部分，一樣要重新編譯核心再重新開機的動作。一旦在核心補釘確定之後，就很難在執行工具時動態去調整想要監看的內容。

- 使用 printk 來輸出想要的資訊對系統本身而言是個嚴重的負擔(overhead)，因為 printk 本身必須牽扯到系統輸出入(system I/O)，在核心處理網路通訊協定堆疊時的速度是相當快的，如果在核心裡面的每個重要函式都加了 printk 勢必對整體的效能(performance)有著顯著的影響。

下一個章節將開始介紹本論文所提出的系統架構，除了達到前述系統目標之外，也會一一地改善 Naive approach 所遇到的種種缺失，以期能完成一個有效率且又能夠動態調整監看內容的平台工具。

4.2 系統架構

Figure 4-3 是本論文所提出的系統架構圖：

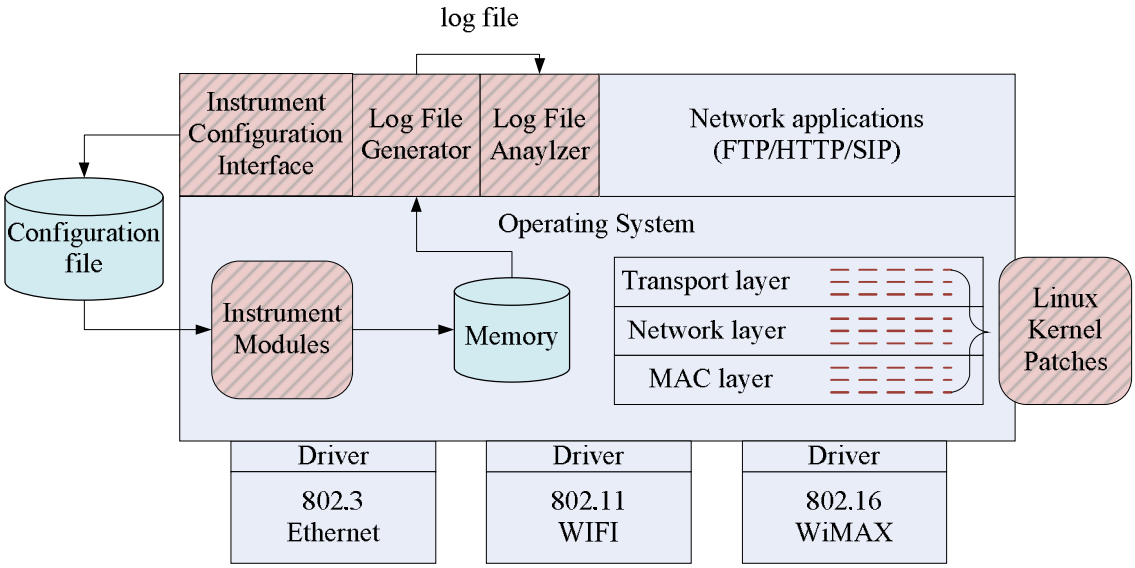

Figure 4-3 論文系統架構圖

圖中主要有兩大部分：(1)淺藍色部分為原始作業系統所擁有的部分，包括了中間區域的作業系統(Operating System)以及其中的網路通訊協定堆疊(Protocol stack，包括 Transport, Network, MAC layer)之外，還有下層對應的網路介面(Network interface，此舉例包括 Ethernet, WIFI, WiMAX)及驅動程式(Driver)，以及上層的網路應用程式(Network applications，此舉例包括 FTP, HTTP, SIP 通訊協定)；(2)斜線的深色部分為本論文新增的主要元件，包括分布在核心網路堆疊裡的虛線區域，此為核心補釘 Linux Kernel Patching 之程式碼，而在作業系統內部的左方還有 Instrument Modules，此部分是用來控制核心補釘的主要模組，主要功能是用來輸出想要記錄的重要資訊，此部分可先參考第二章 2.2Linux 核心模

組的介紹，在之後的章節會更詳細地描述。此外，在作業系統之上還有兩個主要元件，分別是 Instrument Configuration Interface 及 Log File Generator and Analyzer，前者是用來設定 Instrument Modules 所要監看的核心函式，後者是負責輸出 log 檔案以及分析 log 檔案的主要元件。

在作業系統(含)以下的區域都是屬於 kernel space，所有的行為都是受到核心所保護的，所以我們要輸出 kernel space 的系統資訊唯有透過重新修改 Linux kernel 原始碼或是透過系統呼叫(System call)等方法才有辦法達到，詳細資訊可參考[28][29]。然而，所有對 Linux kernel 原始碼的修改都必須是十分謹慎的，因為小小的程式錯誤都有可能造成作業系統當機甚至損毀系統，所以本論文在修改 Linux kernel 原始碼的最大原則是以最小的修改、以不改變原始作業系統的運作模式為主要目標。Figure 4-3 的系統架構圖可以看出會影響 Linux kernel 行為的部分僅包括核心補釘以及 Instrument Modules 兩部分，而其中核心補釘只是在核心的每一個重要函式安插一小段程式碼將控制權移交給 Instrument Modules，而 Instrument Modules 實際上也僅僅是將想了解的資訊記錄到 log 檔案中，完全不會影響 Linux kernel 處理網路通訊協定堆疊的運作。

4.3 系統元件

本節將會詳細介紹論文系統架構的四個重要元件：

- Linux Kernel Patching
- Instrument Modules
- Instrument Configuration Interface
- Log File Generator and Analyzer

以下幾個小節將會詳細介紹各個元件的設計理念，以及如何利用這些元件來達到系統目標，並設法解決 Figure 4-1 所遇到的種種缺失。

4.3.1 Linux Kernel Patching

Linux kernel patching 主要有兩個目的，分述如下：

第一個目的主要是記錄了我們針對 Linux kernel 所修改了哪些部分，藉由 Linux kernel patch 檔案的紀錄，我們可以很快地看出原始的 Linux 核心與修改過後的 Linux 核心之差異處，此差異處即顯示了本論文針對 Linux 網路通訊協定堆

疊的哪些重要函式安插了探針(probe)的動作[30][31]，此探針的行為將在之後做介紹。

第二個目的主要是為了讓修改 Linux 核心的變更內容迅速套用到全新的核心系統上，這對在發展嵌入式系統的網路設備上尤其重要，發展者不可能花許多時間針對每一個相同版本的核心一一植入探針，所以我們可以使用一個 Linux kernel patch 檔案來達到迅速更新的動作。此外，這樣做的好處在於我們只需要在執行完 Linux kernel patching 之後，重新編譯一次核心即可，因為 patch 檔案會將所有可能觀察到的網路核心重要函式都插入了探針，並將其控制權交移給 Instrument modules，可以避免每次對核心網路函式有異動時便要重新編譯核心的動作，在下一段會描述探針與 Instrument modules 的主要關係。

在 4.1 章節中所提到的 Naive approach 中，都是直接在核心的原始碼中插入輸出訊息的程式碼，缺點是在於想要變更輸出訊息則必須重新修改核心，也必須重新編譯核心；而本論文使用探針(probe)這樣的形式就是用來避免這樣的問題，探針實際上並不是直接插入要輸出訊息的程式碼，而是以一個控制權轉移的方式，將真正的要輸出訊息的程式碼放在 Instrument modules 裡面，由 Instrument modules 來控制哪些核心網路函式需要監看以及輸出什麼資訊，示意圖可見 Figure 4-4：

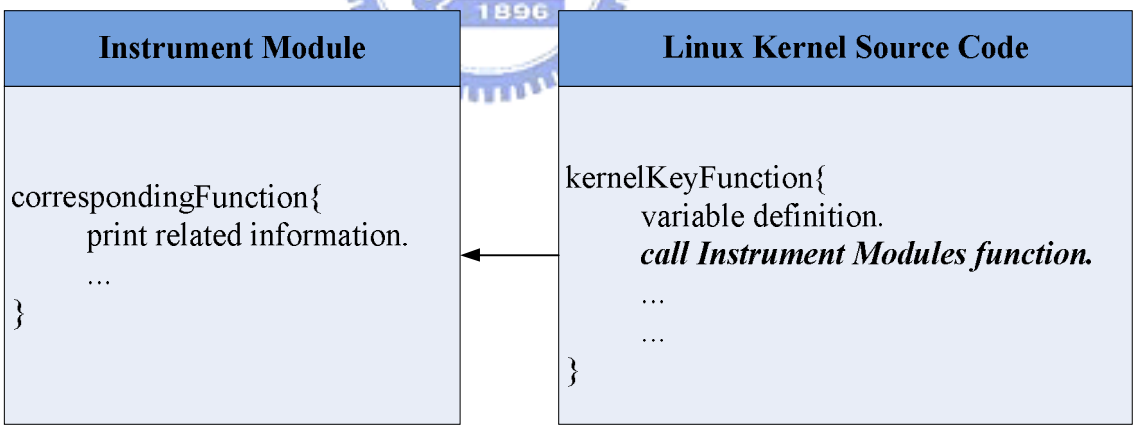

Figure 4-4 Linux kernel patching 之探針示意圖

而 Linux kernel patch 檔案將會把所有可能會監看的核心網路重要函式都以探針的形式插入 Linux Kernel 原始碼中，而由 Instrument Module 來控制要輸出什麼訊息，甚至不做任何動作(即不會監看此函式)，如此一來，執行完 Linux kernel patching 之後只要重新編譯核心一次，此後若要修改輸出的內容或是改變要監看的函式都可以在 Instrument Modules 變更即可，而不需再重新編譯整個核心。

關於 Linux kernel patch 檔案的製作以及套用可以利用 GNU projects 所提供

的 diff 以及 patch 兩個工具[32]來達成，使用 diff tool 來比較兩個 Linux 作業系統原始碼目錄的不同並記錄成一個 patch 檔案，再使用此 patch 檔案用 patch tool 來套用到全新的作業系統上。

4.3.2 Instrument Modules

Instrument modules 是根據 Linux 作業系統所使用的 kernel modules 的方式來撰寫的，有關 Linux kernel modules 可先參考 2.2 Linux 核心模組的介紹。而 Instrument modules 這個元件接續前一節所述，可以用來控制每一個核心網路重要函式的輸出資訊。然而，Instrument modules 設計的目的不僅僅如此，當初 Instrument modules 是為了達到可以動態選擇哪些核心網路函式需要被執行插入輸出指令的效果，舉個例來說：假設發展者只想觀察網路層(Network layer)的相關重要函式，此時可以利用 Instrument modules 在初始化的時候，選擇只輸出網路層重要函式的相關資訊，而對於傳輸層(Transport layer)以及鏈結層(MAC layer)則有如沒有插入任何輸出的程式碼一般，示意圖見 Figure 4-5。

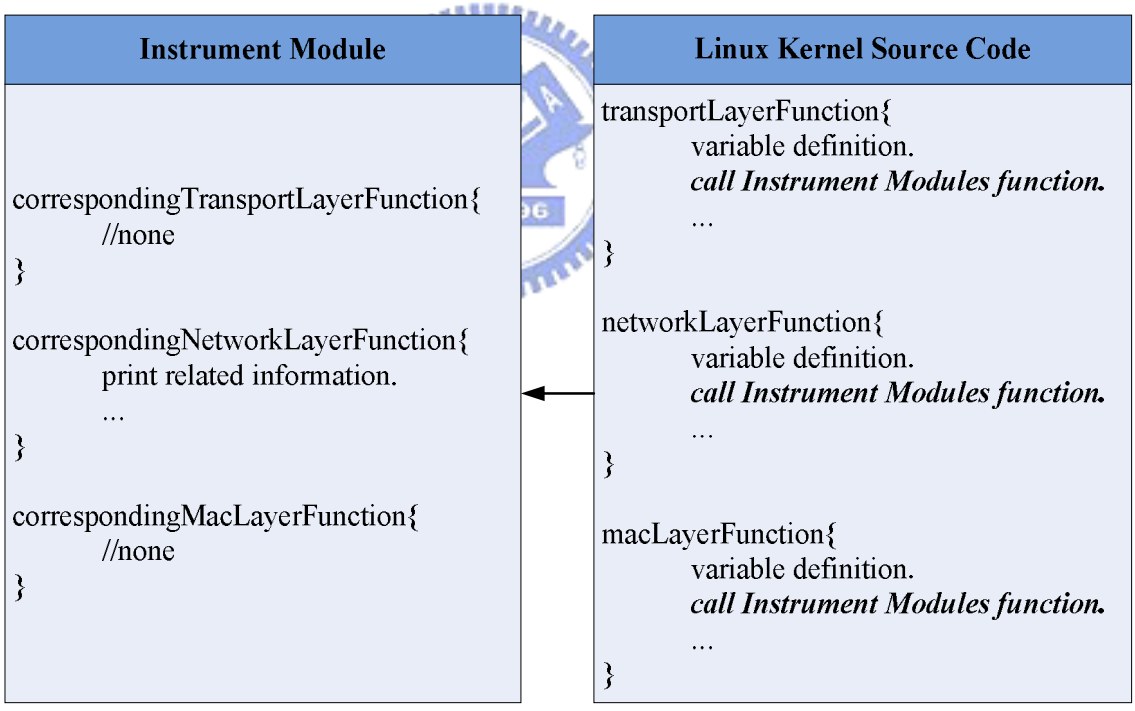

Figure 4-5 使用 Instrument module 選擇網路層函式之示意圖

由於 Linux kernel modules 可以在 user space 中進行編譯，並以掛載(insert module)及卸載(remove module)的方式延伸或補強原本核心的功能，且執行在 kernel space 的模式下，所以才稱 Linux kernel modules 是 Linux kernel 的一個延伸體(extension)。此特點最大的優點有兩項：

- 可以在 user space 以掛載及卸載的方式決定是否要執行此模組的功能，

而不需要一開始就把所有功能都寫死在 Linux kernel 中。對於本論文的工具而言，可以在要執行分析封包的時候才選擇掛載對應的模組，而不是使用修改過的 Linux kernel 來重新開機，同樣都可以達到修改 Linux kernel 功能的目的。

- 當需要修改核心網路函式輸出的程式碼，我們可以直接修改 Instrument modules 的程式碼而間接地達到修改 Linux kernel 原始碼，此時只要在 user space 之下重新編譯模組而重新掛載此模組即可。除此之外，我們也可以利用這項特點達到如 Figure 4-5 的效果，動態地調整想要觀察的核心網路重要函式。

然而，若是以 Figure 4-5 的方式，每次都手動調整哪些函式需要監看、哪些函式不需要監看，對於使用者而言實在不是個明智的方法。由於目前在論文中所有函式的輸出資訊皆是以時間戳記(time stamp)為主，所以基本上可以統一所有函式的輸出內容格式，也因此本論文可以以一個設定檔案(configuration file)來儲存所有要監看的核心網路重要函式名稱，並對每一個函式設定是否需要監看並輸出相關時間戳記的訊息。

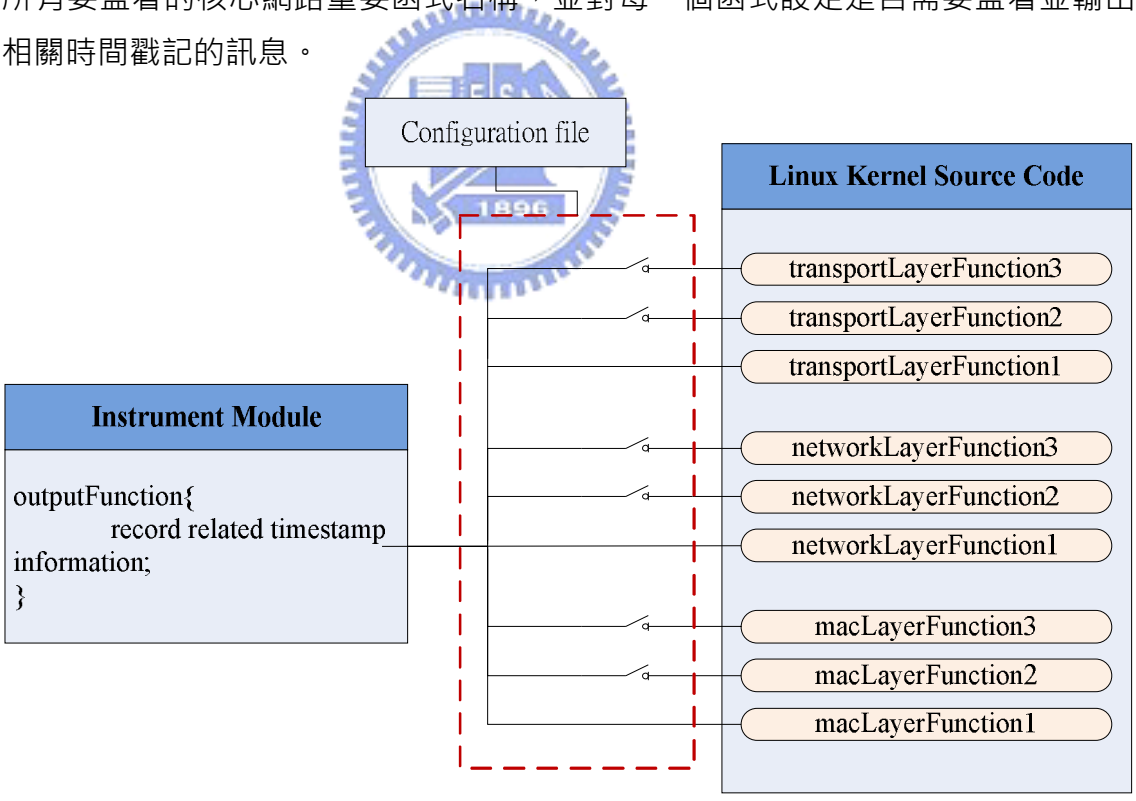

Figure 4-6 Instrument module 使用設定檔案決定監看函式示意圖

如 Figure 4-6 所示，在 Instrument module 讀入了設定檔案之後，便會自動針對每一個網路函式做設定，由於 4.3.1 Linux Kernel Patching 會對每一個重要的網路函式都設定探針(probe)，所以結果就好像示意圖中每個函式都拉出一條線

並接上一個開關(switch)連接到 Instrument module，如果在設定檔案(configuration file)中有被設定監看的話，就會接通(switch on)到 Instrument module 中的訊息輸出函式，圖中的例子是針對網路堆疊中每一層的第一個入口函式(entry function)都做了監看的動作。

談到Instrument modules 記錄輸出訊息的方式，若使用 Figure 4-1 以 printk 的方式直接輸出到/var/log/message，仍然會因為在傳送或接收封包時大量的系統輸出入(system I/O)造成整個網路通訊協定堆疊的負擔，而導致作業系統在處理網路功能時反應變慢。為了解決這樣的問題，在目前的研究中有提出了一些相關的問題與解決方法[33]，而要降低網路通訊協定堆疊的負擔(overhead)，首要條件絕對是要避免使用大量的系統輸出入。

使用記憶體的方式作為儲存是一個解決系統負擔的辦法，本論文使用 proc filesystem 的方式，可先參考 2.3 章節 procfs (/proc filesystem)的介紹，將所有要輸出的 log 資訊以 proc 檔案系統的形式儲存下來。由於 proc filesystem 所儲存的內容相當於直接利用系統的記憶體，所以寫入到 proc 所開啟的檔案下就如同直接將資料寫入記憶體中，於是便避免了使用 printk 這樣過量使用系統輸出入而造成的額外延遲時間。

4.3.3 Instrument Configuration Interface

本節所要介紹的 Instrument Configuration 其實就是針對 4.3.2 Instrument Modules 所提到的設定檔案(configuration file)做設定的動作。本元件提供一個使用者操作介面(user interface)來對這個設定檔案做設定，而 Instrument modules 將會再針對這個設定檔案來選取需要監看的核心網路函式。整體的示意圖可見 Figure 4-7：

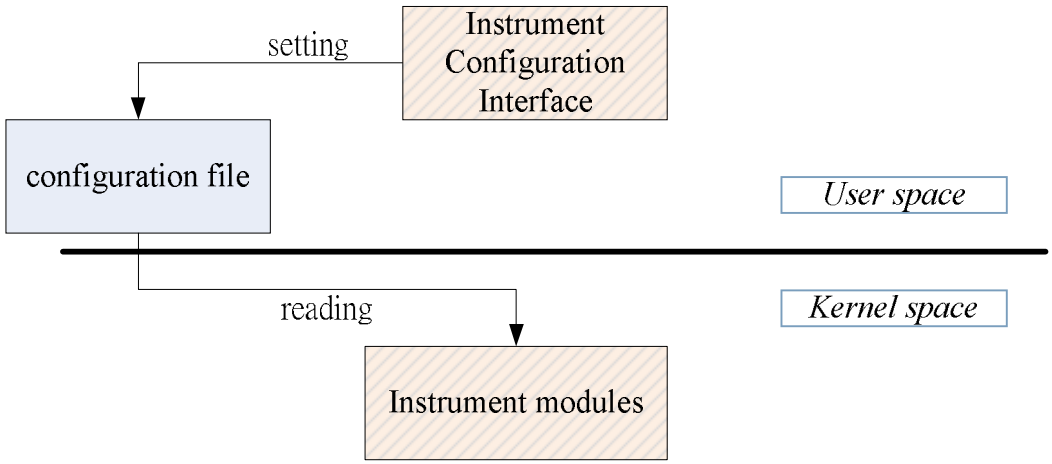

Figure 4-7 Instrument Configuration Interface 與 Instrument modules 之關係

由上圖與 Figure 4-3 的系統架構圖也可了解 Instrument Configuration Interface 是位於作業系統之上，屬於使用者空間(user space)的範圍；而 Instrument modules 位於核心空間(kernel space)中，由讀取設定檔案(configuration file)來決定要監看的核心網路函式。

4.3.4 Log File Generator and Analyzer

由 4.3.2 Instrument Modules 章節可以知道本論文所用來記錄 log 檔案的方式是使用直接記憶體的 proc filesystem 來處理輸出資訊，好處是可以避免使用 printk 的方式而導致過度利用系統輸出入(system I/O)所造成的額外負擔。然而，由於使用 proc filesystem 所記錄的所有資訊都放在記憶體中，並沒有實際以檔案的方式存在檔案系統中，所以本論文在使用完此追蹤工具之後，會將 Instrument modules 儲存在 proc filesystem 中的資訊再轉存成一個 log 檔案放在真實的檔案系統中，負責此工作的元件便是 Log File Generator。

Log File Generator 的另一個好處是所有的工作都是在「離線(off-line)」的狀態下進行，亦即是說對所有的核心網路函式記錄的資訊並不會馬上進行處理，而是先存在記憶體中，等到關閉了監看功能之後再執行 Log File Generator 將所有的資訊存成 log 檔案。如此一來，所有監看核心網路函式的動作都可以很精準地被記錄下來，而不會受到 Log File Generator 一邊記錄一邊使用系統輸出入的干擾，此對監看網路通訊協定堆疊所需的高精確度而言是非常重要的。

而 Log File Analyzer 在本篇論文中僅將此 log 檔案以一個較容易閱讀的方式將結果呈現出來，讓使用者可以很清楚地看出每一個封包在此網路通訊協定堆疊中的流向，以及經過每個重要函式的時間點，藉以了解整個網路行為並對網路延遲的時間做一些假想猜測。然而，智慧型的診斷功能並不在本論文的討論之中，將列為未來工作中之項目。

4.4 總結

在本論文所使用的系統架構下，運用 Linux Kernel Patching 的探針結合 Instrument Modules 的技巧避開了在 Naive approach 所遇到的種種問題，包括了對監看核心網路函式的修改以及輸出訊息的修改都必須要重新編譯核心並重新啟動以載入新核心的麻煩。此外，也運用了 Instrument Modules 與 configuration file 的方式來達到動態的指令嵌入功能，讓使用者可以自行選擇所要監看的函式再執行插入程式碼的效果。最後，使用 proc filesystem 的技巧以達到記憶體直接記

錄 log 檔案的方式，如此一來也避免了 Naive approach 所提到過度使用系統輸出入(system I/O)所造成系統額外的負擔。

第五章 核心網路追蹤工具之實作

5.1 開發環境需求

以下為本論文工具開發實作所使用的環境：

系統核心：Linux kernel 2.6.17

發行套件：Fedora Core 6

編譯工具：GNU GCC 4.1

開機程式：GRUB 1.94

5.2 Linux Kernel Patching 實作

5.2.1 探針插入

此部分將會針對 Linux 網路通訊協定堆疊中每個重要的函式都加入探針(probe)，以交給之後的 Instrument Modules 來執行主要功能。安插探針的方式將以下面的範例來說明：

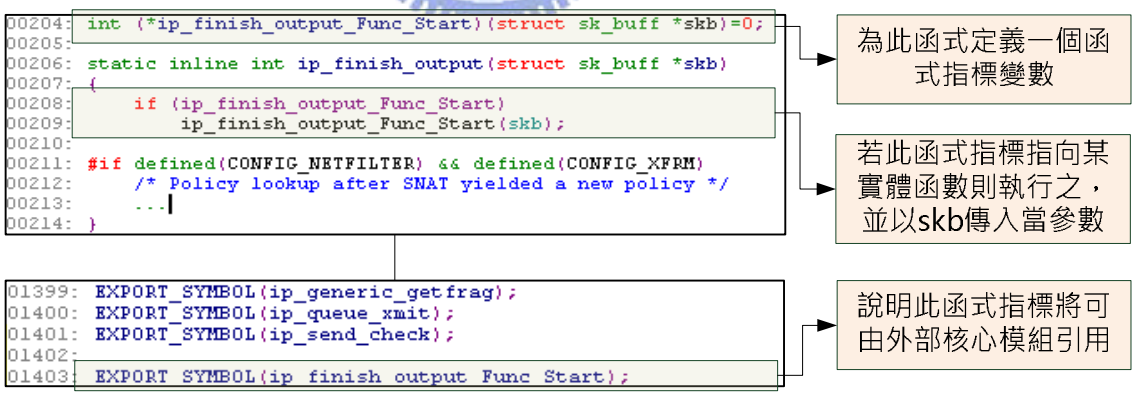

Figure 5-1 Linux Kernel Patching 範例

Figure 5-1 將以 ip_finish_output 函式(位於 net/ipv4/ip_output.c)作為範例說明，而其他函式使用安插探針的方式也相同。一開始先定義一個函式指標變數，即 ip_finish_output_Func_Start，當核心一但執行到這個函式的時候，便會先檢查此函式指標值是否為零，即是否有指向某個函數(定義在 Instrument Modules 中)。若沒有定義此函數指標時便會直接跳過此部分，而繼續執行原始網路通訊堆疊的功能；而倘若 Instrument Modules 有對此函數指標作設定的話，便會用此

函式指標呼叫 Instrument Modules 裡的對應函數，此即 4.3.1 部分所述之控制權轉移。

5.2.2 核心網路通訊協定之函式總覽

本節將列出本論文所有重要且具關鍵性的核心網路函式，將分做傳送端以及接收端兩方面作探討，主要是針對 TCP、UDP 以及 ICMP 通訊協定為主。而詳細的函式內容，如各個函式的參數以及運作原理將不在本論文中詳加解釋，如需詳情可參考[34][35]。

Table 5-1 將列出核心網路堆疊中在傳送端部分的重要函式，將從第四層傳送層往下到第二層 MAC 層：

Table 5-1 核心網路通訊協定堆疊之傳送端函式

函式名稱	檔案位置	所屬通訊協定
inet_sendmsg	net/ipv4/af_inet.c	INET
tcp_sendmsg	net/ipv4/tcp.c	TCP
tcp_push_one	net/ipv4/tcp_output.c	TCP
tcp_write_xmit	net/ipv4/tcp_output.c	TCP
tcp_connect	net/ipv4/tcp_output.c	TCP
tcp_send_ack	net/ipv4/tcp_output.c	TCP
tcp_send_synack	net/ipv4/tcp_output.c	TCP
tcp_retransmit_skb	net/ipv4/tcp_output.c	TCP
tcp_transmit_skb	net/ipv4/tcp_output.c	TCP
udp_sendpage	net/ipv4/udp.c	UDP
udp_sendmsg	net/ipv4/udp.c	UDP
icmp_send	net/ipv4/icmp.c	ICMP

icmp_reply	net/ipv4/icmp.c	ICMP
ip_queue_xmit	net/ipv4/ip_output.c	IP
ip_append_page	net/ipv4/ip_output.c	IP
ip_append_data	net/ipv4/ip_output.c	IP
ip_push_pending_frames	net/ipv4/ip_output.c	IP
ip_output	net/ipv4/ip_output.c	IP
ip_finish_output	net/ipv4/ip_output.c	IP
ip_finish_output2	net/ipv4/ip_output.c	IP
dev_queue_xmit	net/core/dev.c	MAC 層

Table 5-2 將列出核心網路堆疊中在接收端部分的重要函式，將從第二層 MAC 層往上接收到第四層傳送層：

Table 5-2 核心網路通訊協定堆疊之接收端函式

函式名稱	檔案位置	所屬通訊協定
netif_rx	net/core/dev.c	MAC 層
__netif_rx_schedule	net/core/dev.c	MAC 層
net_tx_action	net/core/dev.c	MAC 層
netif_receive_skb	net/core/dev.c	MAC 層
ip_rcv	net/ipv4/ip_input.c	IP
ip_rcv_finish	net/ipv4/ip_input.c	IP
ip_local_deliver	net/ipv4/ip_input.c	IP
ip_local_deliver_finish	net/ipv4/ip_input.c	IP

tcp_v4_rcv	net/ipv4/tcp_ipv4.c	TCP
tcp_v4_do_rcv	net/ipv4/tcp_ipv4.c	TCP
tcp_ack	net/ipv4/tcp_input.c	TCP
tcp_rcv_established	net/ipv4/tcp_input.c	TCP
tcp_recvmsg	net/ipv4/tcp.c	TCP
udp_rcv	net/ipv4/udp.c	UDP
udp_queue_rcv_skb	net/ipv4/udp.c	UDP
udp_encap_rcv	net/ipv4/udp.c	UDP
udp_recvmsg	net/ipv4/udp.c	UDP
icmp_rcv	net/ipv4/icmp.c	ICMP

5.3 動態指令嵌入平台實作

為了可以在使用者空間來控制我們所要觀察的函式，而不需重新編譯整個核心，而達成動態指令嵌入的效果，我們將利用探針插入搭配 Instrument Modules 的方式來完成這項功能。

首先，我們將 Instrument Modules 分做兩個模組，分別是 tx_module 與 rx_module，tx_module 控制傳送端函式部分，而 rx_module 則控制接收端函式部分，由於兩個模組的行為模式大同小異，只是所觀察的函式有所區別，以下皆以 Instrument Modules 代稱。

在 Instrument Modules 的初始化階段，即 init_module 內必須先讀入設定檔案(configuration file)，了解哪些函式是決定被觀察的，以下為一個簡單的設定檔案說明：

```
[1]netif_rx
[0]netif_rx_schedule
[0]__netif_rx_schedule
[1]net_rx_action
[0]process_backlog
[1]ip_rcv
…
[1]tcp_recvmsg
```

Figure 5-2 設定檔案 rx_config.txt

Figure 5-2 是針對於接收端函式的設定檔案，檔案中將會列出所有的函式名稱，並在最前方標示一個數字：1 代表 enabled，表示此函式是要被監看的；0 代表 disabled，表示此函式即使被核心網路執行到也不會有任何額外的處理。

以下圖表為一個 Instrument Module 讀入設定檔案的情形：

```
int init_module(void)
{
        load_configuration("rx_config.txt");
        printk("\n--- Module install ok! ---\n");
        return 0;
}
```

Figure 5-3 rx_module 之 init_module 初始函式

在 Instrument Module 使用 load_configuration 函式讀入設定檔案之後，便會針對「5.2.1 探針插入」的函式指標變數作設定，以下使用虛擬碼來說明 load_configuration 函式的主要工作：

```
If function is enabled                    //probe function
    function_pointer = __function_name;
Elseif function is disabled               //non-probe function
    function_pointer = null;
Endif
```

Figure 5-4 load_configuration 函式之虛擬碼

Figure 5-4 說明了當讀取完設定檔案之後，會針對被設定為 enabled 的核心函式，指定其函式指標的值為" __function_name" 的對應函式，並執行記錄時間以及封包資訊的功能；倘若被設定為 disabled，則會把此核心函式的函式指標設為 NULL，對應到 Figure 5-1 圖中的 if 判斷式時，則會因為此函式指標為 NULL 則直接跳過 Instrument Modules 而繼續執行此核心函式的功能。

以上說明了實作一個動態指令嵌入的流程，以調整設定檔案的方式來達成指令嵌入核心函式的效果，只要將某個核心函式設定起來(enabled)，便好像插入所有要記錄資訊的程式碼到核心網路堆疊中。最大的好處在於嵌入式網路裝置發展者在使用上的方便性，可以在使用者空間中直接調整設定，而不需要每次想要觀察不同的核心函式便要重新編譯核心。

5.4 為封包標示識別碼

由於每個封包在傳送及接收時皆會經過核心網路通訊協定堆疊中許許多多的重要函式，為了區別每個函式是由哪個封包所流經的，以及每個函式之間的關聯性，所以在本論文中我們將會針對網路通訊協定堆疊在上下傳遞時標記特定的識別碼，給予每個封包一個代號。

由於在 Linux 網路堆疊中所用來傳送封包的資料結構主要為第二層及第三層的 struct sk_buff 以及第四層所用的 struct sock，此部分可以由每個函式的參數傳遞中看出，下圖為網路層在接收時傳遞參數的部分程式碼：

```
[net/ipv4/ip_input.c]
int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt, struct net_device *orig_dev)

[net/ipv4/ip_input.c]
static inline int ip_rcv_finish(struct sk_buff *skb)

[include/net/Dst.h]
static inline int dst_input(struct sk_buff *skb)

[net/ipv4/ip_input.c]
Int ip_local_deliver(struct sk_buff *skb)

[net/ipv4/ip_input.c]
static inline int ip_local_deliver_finish(struct sk_buff *skb)

...
```

Figure 5-5 第三層網路接收端以 struct skb_buff 為傳遞參數

sk_buff 可能是 Linux 網路程式碼中最重要的資料結構，用來表示已接收或正要傳輸之資料的標頭。此結構定義於 include/linux/skbuff.h，組成自眾多變數，試著滿足所有網路程式的所有需求。當這個結構從一個層級傳遞至另一個層級時，它的各個欄位也會跟著改變[34]。而 sock 主要是用在第四層傳輸層的資料結構，用來表示每一個 socket 建立時的相關標頭，在第四層傳遞到第三層時會以 sk_buff 中的一個 struct sock 指標變數指向第四層的 sock 標頭，反之亦同。

為了標示每個封包的識別碼，可想而知的，我們必須針對接收端以及傳送端

分別做特別處理。在傳送端，由於我們從第四層往下傳送資料，所以第四層接觸到的資料結構多半以 sock 為主，於是我們針對 struct sock 作一些修改：

```
struct sock {
    ...
    __u32           sk_sndmsg_off;
    int         sk_write_pending;
    void            *sk_security;

    /* For probe purpose */
    unsigned int    probe_flag;
    unsigned int    probe_seq;

    void            (*sk_state_change)(struct sock *sk);
    void            (*sk_data_ready)(struct sock *sk, int bytes);
    void            (*sk_write_space)(struct sock *sk);
    void            (*sk_error_report)(struct sock *sk);
    int         (*sk_backlog_rcv)(struct sock *sk,
                          struct sk_buff *skb);
    void                (*sk_destruct)(struct sock *sk);
};
```

Figure 5-6 修改 sock 資料結構加入識別碼變數

由於我們所觀察的傳送端部分以 TCP、UDP 以及 ICMP 為主，而 TCP 及 UDP 的第一個執行函式皆是由 INET 共通介面之 inet_sendmsg 函式來執行，所以在此函式中我們可以針對上述資料結構之新加入之欄位(probe_flag 及 probe_seq)做設定，ICMP 的部分則是在 icmp_send 及 icmp_reply 函式都要作設定。probe_flag 用來標示是由哪個哪個函式所發出的封包，而 probe_seq 則會在每傳輸一個封包時則給予一個整數值，並在每次傳遞後則遞增。

在接收端的部分，由於封包接收時皆會經過第二層入口函式 netif_rx，(新型的網路介面有些例外，例如使用 NAPI，可參照[34])，所以我們針對於 struct sk_buff 做修改：

```
struct sk_buff {
    /* These two members must be first. */
    struct sk_buff      *next;
    struct sk_buff      *prev;

    struct sock     *sk;
    struct skb_timeval  tstamp;
    struct net_device   *dev;
    struct net_device   *input_dev;

    ...

    /* For probe purpose */
    unsigned int    probe_flag;
    unsigned int    probe_seq;

    /* These elements must be at the end, see alloc_skb() for details.  */
    unsigned int        truesize;
    atomic_t        users;
    unsigned char       *head,
                *data,
                *tail,
                *end;
};
```

Figure 5-7 修改 sk_buff 資料結構加入識別碼變數

其運作原理與傳送端相同，只要在 netif_rx 函式中對這兩個新增變數做初始化動作，之後所流經的每個函式都可以將這些變數記錄下來，如此一來便可以知道每個封包流經各個函式的關係。

第六章 實驗結果與貢獻

6.1 實驗結果分析

本節將介紹為此論文工具所設計的效能分析實驗，包括了使用了 Kernel patching 後對原始 Linux kernel 的網路效能影響以及使用記憶體記錄 log 檔案與 printk 直接輸出 log 檔案的比較，最後，利用此工具來分析 Linux 網路通訊協定堆疊之各層時間所佔之比率。

本論文的實驗環境將有一台 Linux 用戶端主機用來模擬未來之嵌入式網路裝置，並執行本論文的所有程式，主要包括第四章與第五章核心網路追蹤工具之設計與實作，相關硬體與軟體規格表可見 Table 6-1 及 Table 6-2：

Table 6-1 Linux 用戶端主機產品規格表

Category	Specification
PC	ASUS M5A Notebook
Processor	Intel® Pentium® M 處理器 730-770 1.60-2.13GHz, 533MHz, 2MB L2 快取記憶體
Memory	1G (M5Ae)
Processor	筆記型電腦專用硬碟 4200rpm 40GB
WIFI	Z-Com XI-325 802.11b 11Mbps

Table 6-2 Linux 用戶端主機軟體環境

Category	Specification
Operating System	Fedora Core 6
Kernel Version	linux-2.6.17
Development tool	GCC 4.1.1
Testing tools	wget (for HTTP and FTP clients) ping (for ICMP)

此外，實驗將會以有線網路與 802.11b/g 無線網路為主要的測試環境，其他主機包括 FTP 伺服器與 HTTP 伺服器等。

6.1.1 Kernel patching 的 Linux 核心之網路效能測試

本實驗主要是在測量 Linux kernel 進行完 Kernel patching 的動作而尚未開啟 Instrument modules 功能時，針對新的 Linux kernel 之網路功能進行測試，所以本實驗將會比較做完 Kernel patching 後的 Linux kernel 以及原始的 Linux kernel 在不同的網路通訊協定下的網路功能，在此將以傳輸速率與傳輸時間來做比對。

此實驗將分作三個通訊協定來做測試，分別是 FTP、HTTP 及 ICMP，希望得到的結果皆是進行完 Kernel patching 的 Linux 核心在進行三種不同協定的網路傳輸時都能與原始的 Linux 核心一樣順暢。

第一個部分是針對 FTP 通訊協定所做的實驗，分別在有線網路與 802.11b 無線網路環境中下載某一檔案，並重複 15 次取其平均值：

Table 6-3 Kernel patching 網路效能實驗 - FTP 通訊協定

File size: 123208647(Byte)	Kernel Patching 之 Linux kernel	Original Linux kernel
Wired network	Transmission time: 0min48sec Transmission rate: 2508.51KB/sec	Transmission time: 0min48sec Transmission rate: 2500.21KB/sec
Wireless network	Transmission time: 4min50sec Transmission rate: 415.13KB/sec	Transmission time: 4min39sec Transmission rate: 431.8KB/sec

第二個部分是針對 HTTP 通訊協定所做的實驗，分別在有線網路與 802.11b 無線網路環境中下載某一檔案，並重複 15 次取其平均值：

Table 6-4 Kernel patching 網路效能實驗 - HTTP 通訊協定

File size: 68290257 (Byte)	Kernel Patching 之 Linux kernel	Original Linux kernel
Wired network	Transmission time: 0min38sec Transmission rate: 1754.99KB/sec	Transmission time: 0min37sec Transmission rate: 1802.431KB/sec
Wireless network	Transmission time: 2min13sec Transmission rate: 501.67KB/sec	Transmission time: 2min14sec Transmission rate: 496.47KB/sec

第三個部分是 ICMP 通訊協定的測試，一樣是在有線網路與 802.11b 無線網路環境中對某遠端主機(在此以 tw.yahoo.com 台灣奇摩網站的主機為例)進行 ping 的動作來測試網路來回時間(Round Trip Time，RTT)，除去一開始詢問 DNS 所需較長的 RTT 時間之外，取 20 次的來回時間做平均值：

Table 6-5 Kernel patching 網路效能實驗 - ICMP 通訊協定

	Kernel Patching 之 Linux kernel	Original Linux kernel
Wired network	1.152ms	1.118ms
Wireless network	5.158ms	5.223ms

由以上三個實驗可以看出在無線網路環境下的傳輸速率比在有線網路下慢了許多，此外，在無線網路下的傳輸速率也較不穩定，因為在眾多無線基地台同時運作下會有許多的干擾，也因同時連上本實驗所使用的無線基地台之使用者個數，亦會影響所能利用的網路頻寬，所以導致每次測量的傳輸速率變異也較大。本論文已經選擇干擾較少、使用者較少的時間地點來進行實驗，所以所呈現出來的數據已較為平均。

由以上實驗也可以看出執行 Kernel patching 後其實對整體的網路效能幾乎看不出有任何影響。原因其實也不難理解，因為在 Kernel patching 中，實際上只有在每個核心網路的重要函式中加入一個探針而已，而實際上的程式處理片段是放在 Instrument modules 中，故在沒有執行 Instrument modules 的功能下，對整個 Linux kernel 的影響實際上並不大。

6.1.2 procfs 與 printk 輸出 log 資訊之比較

本實驗是針對使用記憶體與直接使用 printk 之系統輸出入方式來記錄 log 資訊做效能分析比較。測試工具是利用’ping’來測試 ICMP 通訊協定下的反應時間，將以網路來回時間(Round Trip Time，RTT)作為衡量指標。

實驗將針對本機端(localhost)網路做 ping 測試，主要原因是因為我們想了解在使用不同機制下記錄 log 檔案所造成的影響，所以若以本機端網路作為來回測試時，由於網路傳輸的反應時間極短，更能由 RTT 時間看出 log 檔案紀錄對整體效能所造成的影響。以下實驗將以 ping 127.0.0.1(localhost)做測試，並取 20 次的來回時間(RTT)平均值做比較：

Table 6-6 使用 system I/O 與 memory 方法記錄 log 檔案之效能比較

	RTT 時間
未記錄任何 log 資訊	0.35ms
System I/O (printk)	1.9ms
Memory (/proc)	0.37ms

由 Table 6-6 可看出在使用 System I/O (printk)的方式下對原始的效能影響甚鉅，相較於使用 memory (/proc)之下儘管也對效能有所影響，但整體對於系統所增加的負擔(overhead)並沒有那麼大。在瀏覽一般網頁時(以 HTTP、TCP 通訊協定為主)，若使用 System I/O 的方式下也會感覺到整體的網路速度降低，因為其中所經過的通訊協定堆疊函式更多，所要記錄的資訊量也更多；相對之下，使用 memory 來紀錄 log 檔案，則可以避免隨時隨地都在開啟檔案、寫入檔案等問題，而減低整體的系統負擔。

然而，memory 的資源也不是無限的，尤其是使用嵌入式的網路設備，其 memory 之硬體資源勢必不多，若測試的時間拉長，log 檔案所要記錄的內容變

多，memory 空間必然會不夠使用。要解決這樣的問題，可能的作法可能包括在每次 log 檔案紀錄即將超過 memory 資源時即輸出至檔案，但此舉仍會影響整體效能；或是於測試階段增加 memory 硬體支援，此部分將不會於本論文中探討。

6.2 貢獻

本篇論文針對 Linux 網路通訊協定堆疊提出了一套高效率資訊記錄以及動態調整觀察點功能的架構，使得嵌入式網路設備開發人員可以快速地了解 Linux 核心在網路部分的整體運作，也可以利用此工具來分析嵌入式網路設備在核心上的效能，進而衡量瓶頸點發生在何處。

此外，本論文使用探針插入(probe insertion)以及核心模組(kernel module)的搭配來達成對核心網路堆疊的動態調整觀察，此架構同樣可以適用於其他有關核心原始碼之修改與偵錯的工具，除了可以加速開發與偵錯中之速度，也可以達成動態調整觀察點之功能。

第七章 結論與未來工作

7.1 結論

隨著網路與電腦科技的進步，嵌入式網路通訊裝置已經逐漸普遍於市面上，然而，目前對於嵌入式網路通訊裝置的相關效能評估工具仍不多，使得我們有必要開發一套合適的輔助工具，協助嵌入式網路通訊裝置的開發者正確界定造成延遲與封包遺漏的原因。

本論文所提出「Linux 網路通訊協定堆疊之高效率動態的指令嵌入平台」便是希望可以讓開發者能夠針對 Linux 網路通訊協定堆疊有更深入的了解，並經由相關通訊協定的測試更了解核心內部的運作；此外，藉由這些測試的資料紀錄，可以用來分析核心內部網路各層的處理時間，進一步分析內部的封包傳輸及接收時間延遲。

本論文提出以核心補丁的探針插入方式搭配核心模組來達到動態調整觀察點的功能，並先利用 memory 的方式來儲存輸出訊息，之後在離線狀態(off-line)將所有資料訊息輸出到 log 檔案中，藉以避免使用過多的 System I/O 而造成系統負擔，以達到高效率的資訊記錄。

在未來中，嵌入式的網路裝置肯定越來越受歡迎，但以現今對於此類裝置之相關測試評量工具仍屬不足。除了本論文所提出的工具之外，未來也希望能夠整合本實驗室學長於 Linux 核心上的相關研究並整合封包過濾器等軟體成為一套整合型的分析工具。

7.2 未來工作

我們的終極目標是希望可以發展一套整合核心系統與通訊行為的網路通訊裝置的評比工具，主要的整合內容重點有三：(1)核心內部通訊協定堆疊之分析(2)核心內部網路事件通知機制(3)封包擷取工具。而本論文已經完成第一部分，而第二部分也由許凱程學長於[23]的論文中實作完畢，綜合(1)、(2)兩部分將完成嵌入式網路通訊裝置的內部(internal)分析，而(3)的部分將於現有的工具來完成外部的(external)分析。因此，未來工作的首要目標便是能夠整合此三大方向，將現有的工具及文獻做整合，完成一套完整的整合分析工具。

另一個部份是智慧型的診斷分析，上述的整合分析工具將可以見到內外部的綜合資訊，包括了內部的核心網路堆疊行為與特定的核心網路事件，以及外部的網路封包及其對應之通訊協定等資訊，但仍需發展一套智慧型診斷演算法來針對這些綜合資訊做進一步的分析，像是如何利用這些資訊數據來判斷此裝置的網路效能瓶頸發生在何處，進而如何針對這些部份做改善的建議等。

Reference

[1] Torvalds Linus and Diamond David, “Just for Fun: The Story of an Accidental Revolutionary”, Harpercollins, Jun. 2002.

[2] Wikipedia – “GNU”, Available from: http://en.wikipedia.org/wiki/GNU

[3] Wikipedia – “Linux”, Available from: http://en.wikipedia.org/wiki/Linux

[4] GNU Operating System, Available from: http://www.gnu.org/

[5] Ubuntu GNU/Linux, Available from: http://www.ubuntu.com/

[6] Fedora Linux, Available from: http://fedoraproject.org/

[7] Bryan Henderson, “Linux Loadable Kernel Module HOWTO”, Available from: http://www.tldp.org/HOWTO/Module-HOWTO/

[8] Peter Jay Salzman, Michael Burian, and Ori Pomerantz, “The Linux Kernel Module Programming Guide”, Dec. 2005.

[9] Doug Abbott , “Linux for Embedded and Real-time Applications, Second Edition”, Newnes, Apr. 2006.

[10] Jongmoo Choi, “Kernel aware module verification for robust reconfigurable operating system”, Journal of Information Science and Engineering, Vol. 23 No. 5, pp. 1339-1347, Sep. 2007.

[11] Chuanxiong Guo and Shaoren Zheng , “Analysis and Evaluation of the TCP/IP Protocol Stack of Linux”, International Conference on Communication Technology Proceedings, vol 1, pp. 444 –453, Aug. 2000.

[12] GDB: The GNU Project Debugger, Available from: http://www.gnu.org/software/gdb/gdb.html

[13] KGDB: Linux Kernel Source Level Debugger, Available from: http://kgdb.linsyssoft.com/

[14] KDB (Built-in Kernel Debugger), Available from: http://oss.sgi.com/projects/kdb/

[15] Linux kernel oops, Available from: http://www.kerneloops.org/

[16] R. Krishnakumar, “Kernel Korner: Kprobes - a Kernel Debugger”, Linux Journal, Jun. 2006.

[17] BPF Filter, Available from: http://www.pcausa.com/support/bpfhelp.htm

[18] W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff, “Unix Network Programming, Volume 1: The Sockets Networking API”, 3rd Edition, Addison-Wesley, Nov. 2003.

[19] Wireshark: The World's Most Popular Network Protocol Analyzer, Available from: *http://www.wireshark.org/*

[20] AiroPeek NX: Wireless LAN analyzer, available from: http://www.airopeek.de/

[21] Sniffer Portable: Network Analyzer, available from: http://www.netscout.com/products/sniffer_portable.asp

[22] Ja-Juan Chou, "Design and Implementation of Driver-Level Network Event Notification Mechanism in Linux", WIN Lab NCTU, 2005.

[23] Kai-Chen Hsu, "Design and Implementation of a Middleware for Network-Aware Applications", WIN Lab NCTU, 2006.

[24] Dave Bakken, "Middleware", Encyclopedia of Distributed Computing, Kluwer, Available from: http://www.eecs.wsu.edu/~bakken/middleware.pdf

[25] J.Salim, H. Khosravi, A. Kleen, A.Kuznetsov, "Linux Netlink as an IP Services Protocol", IETF RFC 3549, July 2003.

[26] GNU GPL(General Public License), Available from: *http://www.gnu.org/copyleft/gpl.html*

[27] Ying-Dar Lin, and Ping-Tsai Tsai, “Trace Linux TCP/IP kernel – Using remote debugging”, Available from: *http://speed.cis.nctu.edu.tw/~ydlin/miscpub/remote_debug.pdf*

[28] Ariane Keller , “Kernel space - user space communication”, Available from: *http://people.ee.ethz.ch/~arkeller/linux/k_u_howto.html*

[29] J. Corbet, A. Rubini, and G. Kroah-Hartman, “Linux Device Drivers, 3rd Edition”, O’Reilly, Feb. 2005.

[30] Moore, R. J., “Dynamic probes and generalized kernel hooks interface for Linux”, Proceedings of the fourth annual Linux showcase and conference. Atlanta, GA, USA, pp. 139-145, 2000.

[31] Nicolas Loriant and Jean-Marc MenaudGeneralized, “Dynamic Probes for the Linux Kernel and Applications with Arachne”, In Proc. of the 2007 IADIS Conference on Applied Computing, Feb. 2007.

[32] GNU diff and patch utilities, Available from: *http://savannah.gnu.org/projects/diffutils/*

[33] Jeong-Won Kim, Young-Uhg Lho, Young-Ju Kim et al, “A memory copy reduction scheme for networked multimedia service in Linux kernel”, Lecture Notes in Computer Science, 2002, 2510: 188–195.

[34] Christian Benvenuti, “Understanding Linux Network Internals”, O'Reilly, Dec. 2005.

[35] Thomas Herbert, “The Linux TCP/IP Stack: Networking for Embedded Systems”, Charles River, May 2004.