國 立 交 通 大 學

資 訊 工 程 學 系

碩 士 論 文

異質網路漫遊系統整合平台之設計與實作

Design and Implementation of an Integration Platform for Heterogeneous Network Roaming Systems

研 究 生：范榮軒

指導教授：曾建超 教授

中 華 民 國 九十四 年 五 月

異質網路漫遊系統整合平台之設計與實作

研究生：范榮軒　　指導教授： 曾建超 博士

國立交通大學

資訊工程學系碩士班

摘要

隨著行動運算技術的普及化，現今市面上多數的可攜式裝置如筆記型電腦、平板電腦、個人數位助理或是智慧型手機等可能皆已具備存取網際網路的能力。 不僅如此，這些裝置通常擁有不止一種的網路介面，包括有線的區域網路、無線的區域網路、GPRS，甚至是 PHS 或是 3G。 然而我們發現到在硬體逐漸成熟的同時，卻缺乏一套可以整合這些異質性網路的軟體讓使用者能根據目前所在的位置選用適當的網路媒體並且維持切換時原有的連線。

在本論文中，我們便設計與實作此異質網路漫遊系統整合平台於裝有微軟視窗作業系統 Windows XP 的可攜式裝置上。 此平台提供的功能有收集網路卡的狀態資訊、自動選擇最適當的網路卡、由選定網路卡繞送/擷取往返的封包以及維持行動裝置原有的連線。 由於網路介面在選擇上必須考量其連線狀態、網路頻寬、計費標準與使用者偏好等複雜的因素，因此本論文僅專注於網路卡管理與路由等方面的技術探討。 為了驗證我們的方法，我們也實作了一個運作於用戶模式下以優先權為考量的切換抉擇演算法軟體模組來為行動裝置自動選擇一張適當的網路卡。

首先，為了選擇一張適當的網路卡，切換抉擇模組必須取得網路卡的相關狀態。 這些狀態是由連結狀態、訊號強度等的鏈結層資訊與諸如 DHCP 位址、DHCP 閘道、路由表與 ARP 快取表等的網路層資訊所共同組成。 其中切換抉擇模組藉由呼叫 IPHLPAPI 函式庫來存取網路層資訊。 但是由網路卡所提供的鏈結層資訊則僅提供給 NDIS 驅動程式來使用。 因此我們便實作了稱為 NdisProt 的核心驅動程式來收集鏈結層資訊並提供給用戶模式下之應用程式存取的介面。 除此之外，為了加速切換，我們在 NdisProt 中也加入了 layer-2 trigger 並且利用 IPHLPAPI 函式庫來攔截 DHCP 事件以減低切換抉擇模組偵測網路卡狀態改變的等待時間。

其次，我們實作了 Mobile IP 來支援行動裝置切換網路卡時的原有連線維持。 在這部分我們所採用的是配置轉交位址模式，如此一來我們便可以視每個網路僅為一個用來存取的網路而不需要網路服務提供者的任何支援。 然而，在配置轉交位址模式下，Mobile IP 軟體必須將應用程式所送出的封包在到達指定的網路卡前做封裝的動作，反之它也要將網路卡所收到的封包在傳遞給應用程式前做拆裝的動作。 因此，為了從 Windows 的 NDIS 架構中攔截不管是由應用程式所送出或是自網路卡所收到的封包做後續的封裝或拆裝，我們設計了兩個核心驅動程式 NdisFlt 與 IpFlt 於此平台中。 此外，由於 Windows XP 在 SP2 之後便不再提供 raw socket 給 Mobile IP 軟體來傳送已封裝/拆裝的封包，Mobile IP 軟體便必須自行做乙太網路訊框的封裝後將該訊框直接送往指定網路卡的迷你連接埠驅動程式。

再者，Mobile IP 需要將家位址固定地設定在行動裝置的其中一張網路卡上，而該網路卡在當行動裝置移動時也需要動態地透過 DHCP 取得配置轉交位址。 所以整合平台運用了 IP Alias 的技巧將家位址與轉交位址同時地設定在一張網路卡上。 不過，在 Windows NDIS 架構中的 Media Sense 功能在網路卡失效時會通知 TCP/IP 協定驅動程式。 一旦 TCP/IP 協定驅動程式收到該訊息，它便會移除該失效網路卡上相關的 IP 位址設定。 因此為了永遠保持家位址的存在，整合平台插入了一隻中間層驅動程式來避免 TCP/IP 協定驅動程式得知設有家位址之網路卡的連結狀態。

最後，整合平台也實作了 IETF 所提出的 UDP 承載 IP 封裝標準讓行動端漫遊於私有網域內。 然而有鑑於 UDP 承載 IP 封裝帶來過多的標頭封裝與封包分割，我們提出了一個新穎的 Mobile IP 穿越網路位址轉譯器機制。由效能評估的結果可發現我們的方法將優於原有 IETF 所提出的。

Design and Implementation of an Integration Platform for Heterogeneous Network Roaming Systems

Student: Jung-Hsuan Fan Advisor: Dr. Chien-Chao Tseng

Department of Computer Science and Information Engineering
National Chiao Tung University

Abstract

As wireless networks and mobile terminal technologies advance, most mobile devices, such as notebooks, tablet PCs, PDAs, or smart phones, may have more than one network interface, including wired LAN, wireless LAN, GPRS, PHS, and even 3G, and can access Internet with any appropriate interface. However, although the hardware of the mobile devices gradually becomes more and more mature, there is still lack of software that can manage these heterogeneous network interfaces and help users adopt the most suitable media to retain the ongoing sessions when mobile devices change the points of network attachment.

In this thesis, we present the design and implementation of a software platform for integrating multiple heterogeneous network interfaces on a mobile device of Microsoft Windows XP. The integration platform provides the functionalities of gathering the statuses of network interfaces, automatically selecting the most appropriate interface, routing/intercepting packets to/from a designated interface and retaining the ongoing sessions for the mobile device. Because interface selection is a complex decision depending on the connectivity, bandwidths and tariffs of the interfaces, and/or sometimes the user preferences, this thesis focuses only on the techniques issues of interface management and routings. For the demonstration purpose, we also implement a priority-based handoff decision software module, which operates in the user space, to select an appropriate interface automatically for the mobile device.

First, in order to select an appropriate interface, the handoff decision module needs to obtain the statuses of the network interfaces. The network statuses consists of layer-2 information, such as link status and signal strength, and layer-3 information, such as DHCP address, DHCP gateway, routing table, and ARP cache. The layer-3 information can be accessed by the handoff decision module by calling the IPHLPAPI library. However the layer-2 information provided by network interfaces can only be accessed by the NDIS drivers. Therefore we implement an

NdisProt kernel driver to collect layer-2 information and provide interfaces for the user level program to acquire the layer-2 information. Furthermore, in order to speedup the handoff process, we also implement layer-2 triggers in NdisProt and hook layer-3 DHCP events with the IPHLPAPI library to reduce the waiting time for the handoff decision module to detect the changes in interface statuses.

Second, we also implement Mobile IP to support session continuity when a mobile device switches from one network interface to another. We adopt the co-located care-of address (Co-CoA) mode of Mobile IP so that our system can treat each network simply as an access network and does not need any supports from the network providers. However, in Co-CoA mode, the mobile IP software needs to encapsulate packets sent by user programs before routing the packets to a designated interface, and decapsulate packets received from an interface before sending the packets to user programs. Therefore, in order to intercept packets, either sent by a user level program or received from a network interface, from Windows NDIS framework for further encapsulation and decapsulation, we also develop two kernel drivers NdisFlt and IpFlt in the platform. Moreover, because Windows XP after the Service Pack 2 does not provide raw socket for the mobile IP software anymore to send the encapsulated/decapsulated packets, the mobile IP software also needs to perform Ethernet frame encapsulation itself and send the frame directly to the miniport driver of the designated interface.

Furthermore, mobile IP needs to bind the home IP address statically to one of the mobile device's interfaces that may acquire Co-CoAs dynamically as the mobile device moves. Therefore the integration platform applies IP alias technique and binds the home IP address and a Care-of Address simultaneously with an interface card of the mobile device. However the Media Sense function of Windows NDIS will notify the TCP/IP protocol driver when an interface becomes inactive. Upon receiving the notification, the TCP/IP protocol driver will remove the IP address configuration of the inactive interface. In order to retain the home IP permanently, the integration platform also inserts an intermediate driver to prevent TCP/IP protocol driver from knowing the link status of the interface bound with the home IP address.

Finally, the integration platform also implements the IETF IP-in-UDP tunnel standard for the mobile nodes to roam under private networks. Nevertheless, because IP-in-UDP tunnel introduces too much header encapsulation and IP fragmentation overhead, we also propose a novel NAT traversal mechanism for mobile IP. The performance results show that our NAT traversal mechanism outperforms the IETF one.

Acknowledgement

本論文得以順利完成首要感謝我的指導教授－曾建超 博士，在過去三年來對於我的耐心指導與多方教誨，尤其是在論文撰寫與口試期間給予不厭其煩的指正與啟發。 同時要向我的論文口試委員：蔡文能 博士與紀光輝 博士致上謝意，感謝他們在百忙中撥冗細心地審查我的論文並提供寶貴的參考意見，使此研究成果表現的臻至完善。 還有要謝謝無線網際網路研究室(Wireless Internet Laboratory)的同學俊儀，學弟大鈞、凱程與宜榮在程式設計的部分不吝嗇地給予協助。而對研究室內其他的學長姊與學弟妹也要感謝他/她們在生活中給我的照顧與陪伴，讓我的碩士生涯豐富而充實。 除此之外也要特別感謝多年來一直關心我的沛沛、廣慧、教昌、佳靜與子貴等好友，在我碰到困難或遭遇挫折之時給我精神上與實質上的協助，讓我順利地克服一次次的難關。最後由衷地感謝在背後默默支持與鼓勵我的父母，有他們的栽培才有今日我的成就。

僅將此成果獻給家人以及所有關心我的師長與親朋好友。

Ode on Intimations of Immortality

What though the radiance which was once so bright
Be now for ever taken from my sight,
Though nothing can bring back the hour
Of splendour in the grass, of glory in the flower;
We will grieve not, rather find
Strength in what remains behind

～ *William Wordsworth*

Contents

Figures

Tables

Chapter 1 Introduction

Section 1.1 Motivation

被稱為 GPRS 之父的芬蘭赫爾辛基科技大學 Hannu Kari 教授在 2001 年 7 月來台的一場演講中指出，由於第三代行動通訊系統的投入成本過高且推展不易，未來的行動數據服務將是以 GPRS 加 WLAN 為主流的解決方案。 雖然 Hannu Kari 所提出論點並不為 3G 業者所認同，不過在 3G 短期內無法落實大量商業化下，為滿足消費者對於隨身攜帶與高頻寬的需求，手機與 WLAN 的結合似乎成為各大半導體業者的努力方向。 無論是檢視市面上現有的消費性產品或是政府與業界所大力推動的雙網整合計畫，都在在地證實 Hannu Kari 教授的論點。

就技術而言，GPRS 的傳輸距離可達 10 公里，理論速度為 144Kbps。 而相較於 WLAN 傳輸速率高，如 802.11g 可達 54Mbps，但傳輸距離短只有數百公尺。 除此之外，WLAN 採用免費的 2.4GHz 頻段，整體建置成本低，因此收費低廉。 而 GRRS 雖然網路成本高收費貴，但範圍廣且安全性高。 由此可看出，此兩異質性的網路各有其優缺點與適用的空間。 倘若能加以整合便能讓使用者根據需求及環境選用最適當的網路，這也就是驅使我們整合兩大技術的原因所在。

順著該趨勢的發展，現下無論是筆記型電腦，平板電腦，個人數位助理或是智慧型手機在硬體上都已具備雙網甚至是多網的能力。 然而令人失望的是雖然硬體已趨成熟，但是仍未見到一套可統整使用這些異質網路介面卡的軟體。 於是乎便激發了我們發展異質網路漫遊系統的整合平台，藉由該平台希望能夠讓使用者輕易的管理與使用其擁有的多張網路卡，不僅如此，我們還期望保障使用者在這些異質網路間漫遊(切換)時不造成斷線。

Section 1.2 Approach

在上一節中，我們已經清楚的瞭解我們的需求所在，因此在本節中將簡單的敘述我們如何達該目的。 首先，在開發平台的部分，我們將選用被現今一般使用者所最為熟悉的微軟作業系統 Windows XP。 而此平台將提供的功能則包含有收集網路卡的狀態資訊、自動選擇最適當的網路卡、由選定網路卡繞送/擷取往返的封包以及維持行動裝置於漫遊時的連線等。

接著在細部的考量上，我們發現到網路介面的選擇上由於必須考量其連線狀態、網路頻寬、計費標準與使用者偏好等複雜的因素，因此我們僅將重心專注於網路卡管理與路由等方面的技術探討。 此外，驗證我們的方法，我們將會實作一個運作於用戶模式下的簡單切換抉擇演算法軟體模組來為行動裝置自動選擇一張適當的網路卡。 為了做出正確的選擇，切換抉擇模組必須取得網路卡的相關狀態。 而這些資訊包含有 Layer-2 的連結狀態、訊號強度等以及 Layer-3 的 DHCP 位址、路由表等。 所以在這部分將會去瞭解如何自 Windows XP 的網路核心架構中取得這些資訊。

其次，在行動管理的部分我們會採用行動網際網路協定的配置轉交位址模式來支援不斷線的漫遊服務。 採用該模式的原因在於行動端可以視每個漫遊的網路僅為一個存取網路，而無須網路服務提供者的任何支援。 不過，在此碰到的第一個問題就是如何解決該協定中所規定的行動端必須自行處理收送封包的封裝/拆裝動作，換句話說，我們要先能夠攔截原本自應用程式所發送以及由網路卡所接收的封包後才能進行後續的動作。 針對這個問題，我們會去瞭解 Windows XP 內部的網路核心架構尋求可能的解決方案。 第二個問題則是如何管理行動端的家位址與配置轉交位址。 關於這點我們則是希望能讓家位址設定在行動端的其中一張網路卡上，並且永遠維持該設定的存在，同時該網路卡還能透過 DHCP 協定去取得漫遊網路當地的配置轉交位址。

另一方面，在採用行動網際網路協定的同時還會碰到的一個問題就是該協定無法穿越現今普遍存在於網路上的網路位址轉譯器。 對於該問題在 IETF 所提出的解決方案中是將封裝的層級拉到傳輸層來克服，但我們認為這樣有違行動網際網路協定在最初的設計理念。 因此，在最後的部分我們會試著回到網路層重新地解決該問題，並證實我們所提出的方法比原有的解決方案具備更佳的傳輸效能。

最後，在建立起以上這些背景知識後，我們會將所需要的功能做成一個個小的軟體元件，最後以元件組合式的架構建構出成我們所需要的平台。

Section 1.3 Synopsis

本篇論文在稍後的章節編排與簡述如下： 第二章為背景知識的概述，包含行動網際網路協定，網路位址轉譯器以及微軟視窗作業系統下的封包路由與擷取技術。 第三章接著介紹與本論文相關的文獻，其中有無線網路整合技術與行動網際網路協定穿越網路位址轉譯器方法等兩部分。第四章便說明我們所開發的整合平台之系統架構與其內各個軟體元件。 第五章則是我們所提出用來改進原有行動網際網路協定穿越網路位址轉譯器的方法介紹。 緊接著，第六章將詳細地解說如何實作出以上兩個章節所介紹的系統與方法。 並在以實際程式所展示的實例之後，於第七章做個討論與效能評估。 最後，第八章為本篇論文的總結與未來的研究方向。

Chapter 2 Background Material

Section 2.1 Mobile IP

在 90 年代初期，網際網路工程工作特別小組(Internet Engineering Task Force，IETF)於網際網路領域(Internet Area)下創立了一個名為"IP Routing for Wireless/Mobile Hosts (Mobile IP)"的新工作群組(Working Group)，該群組將發展一套行動性的網際網路協定(IP Mobility)機制使得行動用戶(Mobile User)無論在何處連上網際網路皆可使用相同的網際網路位址。 其目的是當行動用戶漫遊於不同的網路時能保持在線狀態並維持原有的網際網路連線(IP Connectivity)。 而這些不同的網路是可以基於不相同的存取技術(Access Technology)，因此這項機制必須獨立於其下的鏈結層(Link Layer)技術。

第一版的規格 RFC 2002 "IP Mobility Support" [8] 於 1996 年的十月被發表，之中提出了一個稱為"Mobile IP for IPv4"的新協定，該協定實際上是正規網際網路協定的一種延伸(Extension)。在 2002 年的一月，新版的規格 RFC 3220 "IP Mobility Support for IPv4" [15] 淘汰了舊有的 RFC 2002，而隨後在八月所發表的 RFC 3344 "IP Mobility Support for IPv4" [16] 也翻新了 RFC 3220。 RFC 3344 目前的狀態為"被提議的標準(Proposed Standard)"，意味著它將進入成為"網際網路標準(Internet Standard)"的程序，但由於仍未完全成熟，將來也有可能更進一步的發展。在此同時，該工作群組也致力於解決一些偶然碰到的特殊問題。 在他們的網站中可以找到發佈於草案(Draft)內的建議解決方案。 其中有些問題將在稍後的章節作討論。

而在 IPv6 方面，工作群組發展了一套類似的機制，該機制被發佈於名為"Mobility Support in IPv6"的草案內。 最主要的差異點在於它將會被整合進 IPv6 的標準，而非如同 IPv4 一般，行動性的支援(Mobility Support)只是選擇性的。 對於行動用戶來說，他將需要特殊的客戶端軟體來使用 Mobile IP。

行動性的支援其需求存在於那些被網際網路用戶漫遊於不同的網路下所使用的服務內。 例如你正在下載一個大的檔案或是正在看或聽一個媒體串流，你將不希望因為在不同的網路之中移動造成這個連線被中斷。 同樣情況也發生在當你使用網際網路語音(Voice-over-IP，VoIP)時，你不會想要每換一次網路就再次地重新建立你的通話。 另一個例子是當行動端(Mobile Node)作為伺服器時，必須用相同的網際網路位址被它的客戶端所找到。 因此，在用途上大致可分為兩基本面：

1. 可利用性(availability)，行動伺服器必須被一個永不改變的網際網路位址所找到
2. 連續性(continuity)，當變更網路的接點(point of attachment)時維持網際網路的連線

Mobile IP 是一個宏觀移動(macro mobility)的解決方案，它能處理在不同存取技術或相異第三層網路間的網際網路換手。 但它並不適合於第二層基地台(cell)間的換手，也就是所謂的微觀移動(micro mobility)。 Mobile IP 近年來逐漸在市場上產生影響力，其中 cdma2000 社群(cdma2000

是北美版本的第三代行動電話系統，由 3GPP2 組織所制定)是推展建置的其中一個主要原動力。在該規格 [22] 中，Mobile IP 被選為當作 cdma2000 封包資料用戶的行動網路協定。

在本節中將根據 RFC 3344 介紹 Mobile IP 這個協定。 首先 2.1.1 是一個概觀的描述，接著 2.1.2 與 2.1.3 將會詳細解釋細節。 最後一些特殊的問題會在 2.1.4 做討論。

2.1.1 Mechanism

在網際網路協定(Internet Protocol, IP) [2] 下的網路，路由(Routing)是根據固定不動的 IP 位址。 一個網路上的端點(Node)可以被正常的 IP 路由所遞送是根據它在網路上所被分配的 IP 位址。 IP 位址由兩部分所構成： 網路前置碼(Network Prefix)與當地位址(Local Address)。 由分屬於不同網路下的端點 A 送給端點 B 的 IP 封包可以被正確地路由到正確的網路是根據其網路前置碼，也就是 IP 位址的第一部分。 當端點 B 經過移動並經由別於家網域(Home Network)的網路連上網際網路，它可能會得到一個與其家位址(Home Address)不同的 IP 位址。 而 Mobile IP 就是使其他端點都可以用端點 B 原本的家 IP 位址與其連線的機制。 再者，有了 Mobile IP，端點 B 能用它原有的家 IP 位址當作來源位址。 所以說 Mobile IP 為行動端提供了與它們的網際網路接點無關的透明化 IP 路由。

根據 RFC 3344，下述將說明幾項定義，其中包含了 Mobile IP 的三個主要構成要素： 家代理器(Home Agent)，外地代理器(Foreign Agent)與行動端。

- 行動端(Mobile Node, MN)

  行動端是一個漫遊於不同網路下並且可以使用相同 IP 位址的主機(Host)或是路由器(Router)。 行動端可以改變其網際網路的接點而不必更動它固定的 IP 位址，也就是所謂的家位址。

- 家網域(Home Network, HN)

  行動端的家網域是指用家位址作為目的 IP 封包依正常 IP 路由所被送達的網路。

- 家位址(Home Address)

  行動端的家位址是一個分配給行動端的固定 IP 位址。 它是維持不變的並且屬於家網域的網路。 行動端將永遠使用這個位址當作來源位址，甚至是正當它在漫遊時。

- 家代理器(Home Agent, HA)

  行動端的家代理器是一個在家網域的主機或是路由器。 當行動端離開家網域的時候，它攔截所有送往行動端的 IP 數據資料(Datagram)並經過通道技術(Tunnel)將它們傳給行動端。 再者，家代理器還執行某些驗證(Authentication)與管理(Administration)的功能。

- 通訊端(Correspondent Node, CN)

  通訊端是一個行動端正在與之通訊的任意端點。

- 外地網域(Foreign Network, FN)

  外地網域是指行動端的家網域之外的網路。

- 外地代理器(Foreign Agent, FA)

  外地代理器是一個在外地網域的主機或是路由器，它提供路由服務給已註冊的行動端。 外地代理器把家代理器經過通道技術傳送過來的 IP 數據資料進行拆封(Detunnel)並發送給註冊的行動端。 再者，外地代理器為行動端執行驗證與管理的功能。 對於行動端來說，它被當作預設路由器(Default Router)。 外地網域不必要存在一個外地代理器來使得 Mobile IP 可以運行。

- 行動代理器(Mobility Agent)

  行動代理器是一個對於家代理器與外地代理器的通稱。

- 轉交位址(Care-of Address, COA)

  轉交位址是家代理器把數據資料經過通道技術要送給行動端所預定到達的 IP 位址。這個通道的終止點可以是行動端所註冊的外地代理器，或者是當沒有可用的外地代理器時，也可以是行動端本身。 對後者來說，這個轉交位址又被稱為配置轉交位址(Co-located COA)。 它是一個在所拜訪的外地網域上暫時分配給行動端的 IP 位址，例如透過動態主機設定協定(Dynamic Host Configuration Protocol, DHCP) [12]。 然後，行動端將會自行對封裝過的數據資料進行拆封。

- 行動連結資訊(Mobility Binding)

  指的是行動端的家位址，轉交位址與剩餘註冊有效時間(Lifetime)的連結。 行動連結資訊被儲存在家代理器上的註冊列表內。

上述的定義已經透露很多關於 Mobile IP 的機制。 為了更進一步的解釋 Mobile IP 的機制，兩種基本的運作模式範例將在以下作說明。 在這些範例中所使用的 IP 位址並非真實的，純粹為了例證。

■ 外地代理器轉交位址模式 (Foreign Agent Care-Of Address Mode)

此模式描述了一個使用家代理器與外地代理器的標準配置範例。 家網域(140.113.215.0/24)透過路由器(140.113.215.254)連接到網際網路，詳見圖 2-1。 行動端固定的家 IP 位址是 140.113.215.207，而家代理的 IP 位址是 140.113.215.206。

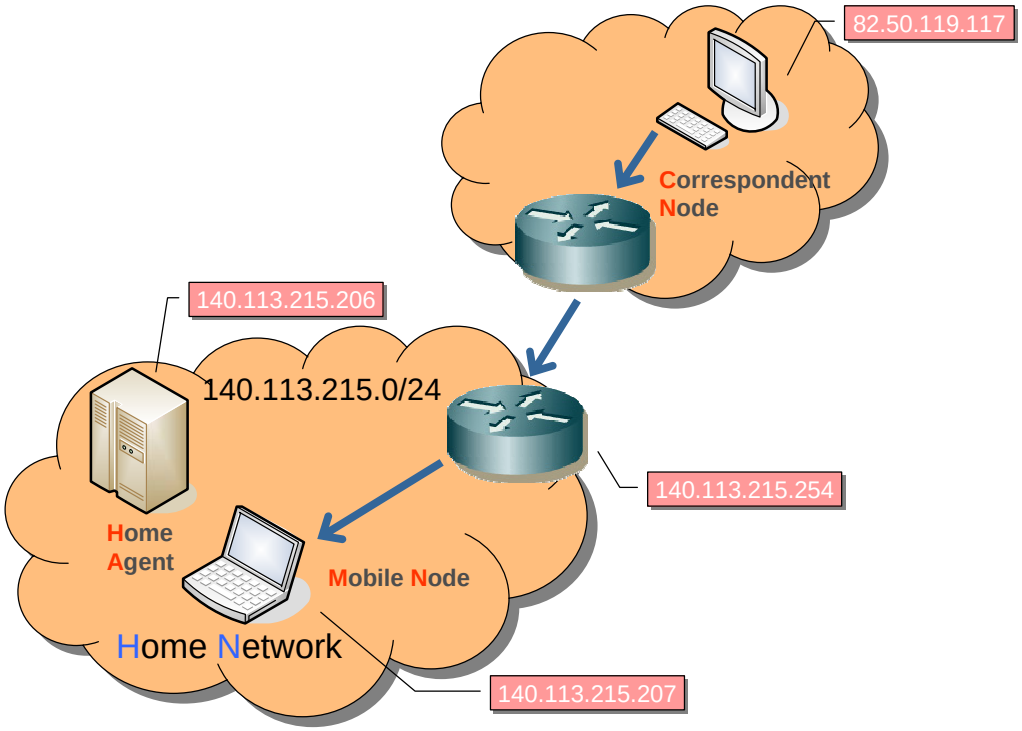

Figure 2 - 1: Home Network Scenario

行動端透過接收來自其家代理器的特定訊息： 家代理器公告(Home Agent Advertisement)，來判斷是否位在家網域內。 大致上來說，家代理器公告是由家代理器所送出，並且帶有家代理器的 IP 位址，有效期限以及家代理器關於 Mobile IP 服務與能力的參數。 當行動端在家網域內時，它能使用正常的 IP 路由與它擁有的家位址。 在這狀況下，家代理器只作為讓行動端判斷所在位置的依據並不干涉其 IP 資料封包的路由。

假設當行動端漫遊到一個外地網域 140.113.24.0/24，如圖 2-2 所示。 行動端可以藉由判斷目前所聆聽的代理器之公告有效期限到期後，是否沒有再收到從同一個代理器來的新公告得知自己的移動。 另一種移動偵測的方式是基於網路前置碼的不同。

行動端藉由接收到外地代理器公告(Foreign Agent Advertisement)訊息發現可用的外地代理器，訊息中提供了一個或多個轉交位址給行動端。 現在行動端得知了外地代理器與它可用的轉交位址，下一步就是透過外地代理器向家代理器註冊。 這個註冊的必要性在於在家代理器上產生一個行動連結資訊，用來控制家代理器與轉交位址間的通道。 該通道將被作為轉交送往行動端的 IP 封包之用。 為了維護這個通道，行動連結資訊必須在註冊有效期限到期前被更換或更新。 Mobile IP 的註冊程序採用要求-回覆(Request-Reply)模式。 假如行動端透過外地代理器註冊，那麼外地代理器將會處理註冊要求訊息(Registration Request Message)然後轉交給家代理器。 當註冊成功，家代理器會回覆給外地代理器，它在轉交回行動端之前會再一次的處理這個註冊回覆訊息(Registration Reply Message)。 所有的註冊訊息都是被經過驗證保護的避免被不必要的干預與濫用。

圖中的箭頭表示了 IP 封包如何在通訊端(82.50.119.117)與行動端間被路由。 實線部分是通訊端送給行動端的封包，虛線的部分是由行動端送給通訊端。 通訊端用行動端的家位址(140.113.215.207)作為 IP 封包送達的地點。 這些封包基於網路前置碼 140.113.215.0 被遞送到行動端的家網域。 在家網域內，家代理器攔截了這些封包並透過通道技術將它們往轉交位址送去，也就是往外地代理器。 外地代理器對這些被封裝過的封包進行拆裝後，把它們丟在行動端所在的鏈結上。 如此一來，當行動端在連接在外地網域的時候就能收到送往它的家位址的 IP 封包。

由行動端送往通訊端的封包可以經由正常的 IP 路由。 行動端用通訊端的 IP 位址(82.50.119.117) 作為封包送達的地點。 它把家位址 140.113.215.207 當作 IP 封包內的來源位址，基本上這樣對網路的拓樸來說是不對的。 此時，外地代理器會作為行動端的預設路由器/閘道器。 這樣的路由方式被稱作三角式路由(Triangular Routing)。 從通訊端到行動端透過通道的路徑不相同於行動端到通訊端的路徑，如此形成了一個三角形。

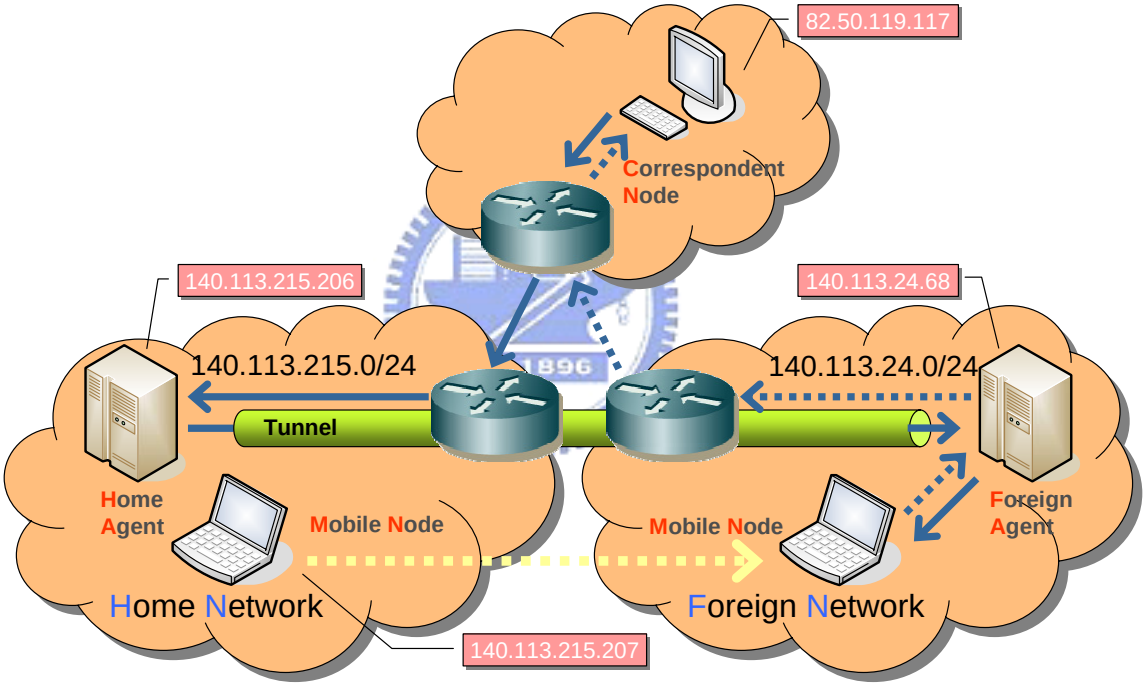

Figure 2 - 2: Foreign Network Scenario under FA CoA Mode

■ 配置轉交位址模式 (Co-located Care-Of Address Mode)

此模式是一個不使用外地代理器的標準範例。 假設行動端連接到外地網域 140.113.24.0/24，見圖 2-3。 這個外地網域沒有設置外地代理器，所以行動端將不會接收到任何外地代理器公告。在這範例下，行動端透過 DHCP 得到一個 IP 位址 140.113.24.23。

行動端能把這個位址當作轉交位址，因為它被分配到行動端的一張介面卡上，這個位址被稱為配置轉交位址。 為了產生行動連結資訊，行動端直接地向家代理器(140.113.215.206)註冊。 家代理器此時便可以建立一條往轉交位址,的通道，在這個例子中其實就是行動端本身。

同樣的，圖中的箭頭表現了一個路由的例子。 與外地代理器轉交位址模式的情況可說是大同小異，除了通道的終點是在行動端之外，也就是說，行動端會自行對封包作拆封。 因此，行動端可以藉由兩個 IP 位址被找到： 140.113.215.207 被用來讓資料穿過通道以及 140.113.24.23 被用作 Mobile IP 的註冊。 而後者是通道的終點。

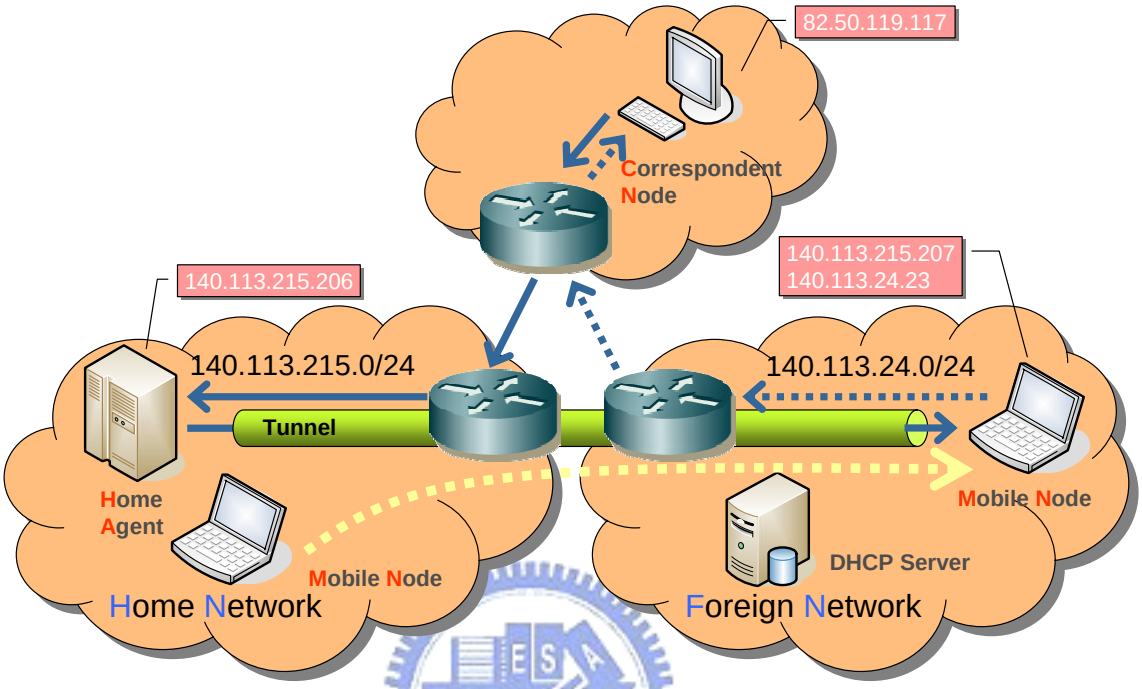

Figure 2 - 3: Foreign Network Scenario under Co-located CoA Mode

2.1.2 Signaling

在 Mobile IP 的協定中定義了幾種特定的訊息，這些訊息主要是為了兩項目的： 尋找行動代理器及其服務與向行動端的家代理器註冊。 在本段中將依序詳細探討關於這兩種訊息。

- 代理器公告與請求 (Agent Advertisements and Solicitations)

在前述的範例中已經提到家代理器與外地代理器使用特定訊息來向鏈結上的所有端點表明它們的存在以及 Mobile IP 的需求與能力。 這些訊息就是所謂的代理器公告。 由家代理器所送出的稱為家代理器公告，而自外地代理器送出的則稱作外地代理器公告。 行動代理器用廣播的方式送出這些訊息，而且只送到它們所服務的鏈結上。 一般來說，每隔一段固定的時間就會送出公告，但這並非是必須的。 因此，代理器公告可被所謂的代理器請求所要求。 當行動端廣播了一個代理器請求訊息到鏈結上，所有在該鏈結上可用的行動代理器都必須回覆一個代理器公告。 在這個情形之下，公告訊息應該被直接送往要求的行動端，也就是並非透過廣播的方式。

代理器公告是由一個行動代理器公告延伸加上一個路由器公告訊息所組成。 路由器公告訊息定義在網際網路控制訊息協定的路由器探索協定(ICMP Router Discovery Protocol, IRDP) [3] 中，圖 2-4。 而行動代理器延伸訊息則包含了關於行動代理器服務的資訊，圖 2-5。 它宣佈了發送的行動代理器是否作為外地代理器，家代理器或者兩者皆是。 它通知了行動端關於它的服務，所支

援的通道模式以及如果是有外地代理器功能的時候，它公告了一個或多個轉交位址。 代理器公告沒有經過驗證或加密。

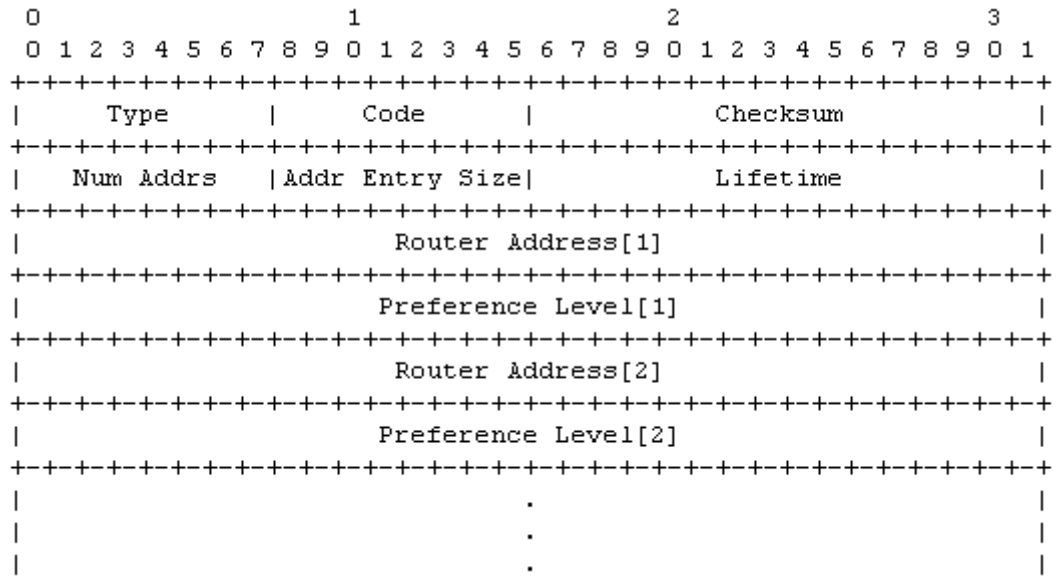

Figure 2 - 4: ICMP Router Advertisement Format

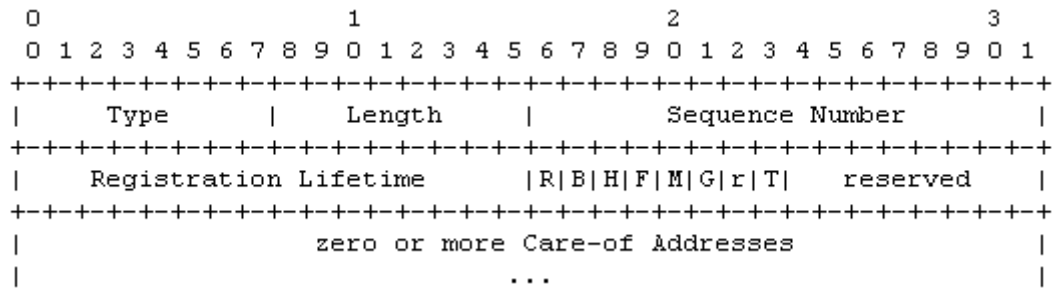

Figure 2 - 5: Agent Advertisement Extension Format

代理器請求訊息就是把 TTL 欄位設為 1 的網際網路控制訊息協定路由器請求訊息(ICMP Router Solicitation)，圖 2-6。 所以它只在被送出的子網域中有效。 此外行動代理器請求訊息中不包含任何特定的 Mobile IP 參數。

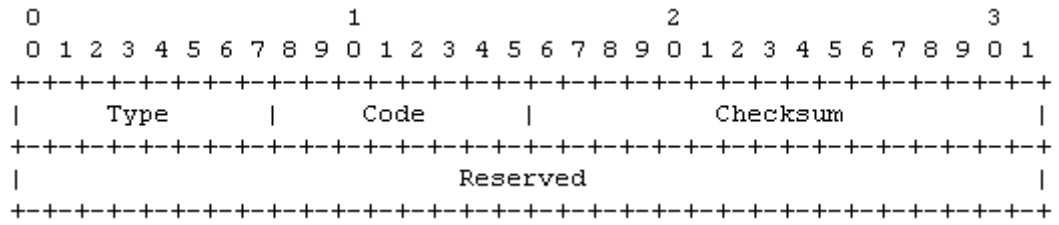

Figure 2 - 6: ICMP Router Solicitation Format

■ 註冊要求與回覆 (Registration Request and Reply)

當行動端漫遊到一個外地網域的時候，Mobile IP 使用了一個通道來重新導向送往行動端的 IP 封包。 這個通道是被家代理器與行動端所建立與控制的，是否與外地代理器連結則是選擇性的。這些控制資訊是藉由註冊訊息來交換。 而 Mobile IP 註冊訊息則是使用用戶數據資料協定 (User

Datagram Protocol, UDP) [1]。 公認的埠號 434 已經被網際網路號分配機構(Internet Assigned Number Authority, IANA)分配作為這樣的用途 [5]。 註冊要求與註冊回覆是由附加一個或多個延伸的標準固定部分所構成，如圖 2-7 與 2-8。

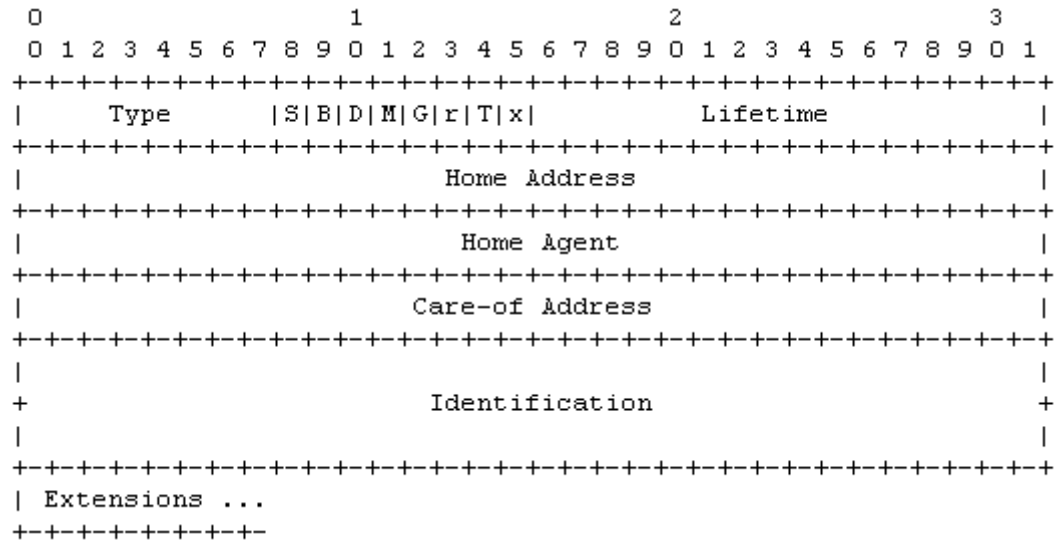

Figure 2 - 7: Registration Request Message Format

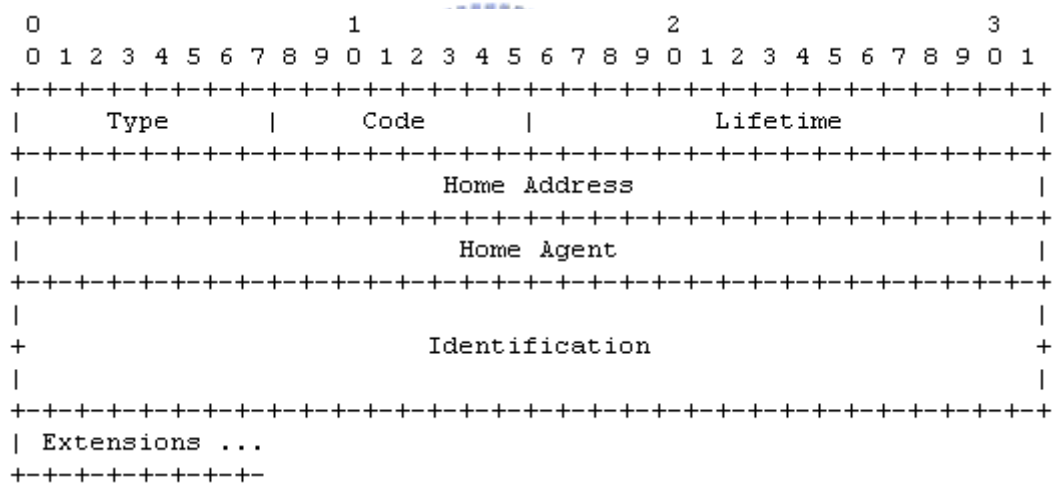

Figure 2 - 8: Registration Reply Message Format

註冊要求與註冊回覆的固定部分只差在一個位元組與一個欄位。 固定部分一定包含了用來識別行動端的行動端家位址，用來識別家代理器的家代理器 IP 位址以及一個用來識別註冊訊息本身的識別欄位，它被用來匹配註冊要求與註冊回覆。 在相異的位元組中，註冊要求包含了行動端可以要求特定 Mobile IP 服務的旗標。 註冊回覆則是在這個位元組中填入家代理器的回覆代碼，它通知了行動端關於這個註冊的狀態。 如此一來行動端便能核對這個註冊是否已經被接受並且在錯誤代碼被回報時採取適當的行動。 而對於這個相異的欄位只存在註冊請求中，它存放了一個轉交位址用來告知家代理器通道的終點。

在 RFC 3344 中定義了許多種類的延伸，其中最重要的就是驗證延伸，見圖 2-9。 這個延伸確保了訊息交換的安全。 每個行動端與它的家代理器間都有一個安全性關聯(Security

Association)。 這個關聯是被一個 32 位元的數字所識別，也就是安全性參數索引(Security Parameter Index, SPI)。 選擇性地，行動端可與外地代理器間有一個安全性關聯，外地代理器也可與家代理器間有安全性關聯，而這些安全性關聯都被各自的 SPI 所表明。 安全性關聯中包含了一個機密，一個 128 位元的共享金鑰(Shared Key)，其被運用在自註冊訊息中的特定欄位計算出一個驗證元(Authenticator)。 驗證元是一個數學上的訊息摘要，根據的是 HMAC-MD5 演算法[11]。 這是一種只讓共享金鑰的擁有者閱讀的簽章方式。 它必須使家代理器與行動端免受來自其他端點的偽造註冊訊息。 同時還有另一種在註冊程序中必須被保護避免的攻擊形式： 重送攻擊(Replay Attack)。 在註冊訊息固定部分中的識別欄位包含了一個時間戳記(Timestamp)或是亂數(Nonce)來確保一個註冊訊息僅有效一次。 這將會避免當惡意的端點重送舊訊息時所造成的困擾。

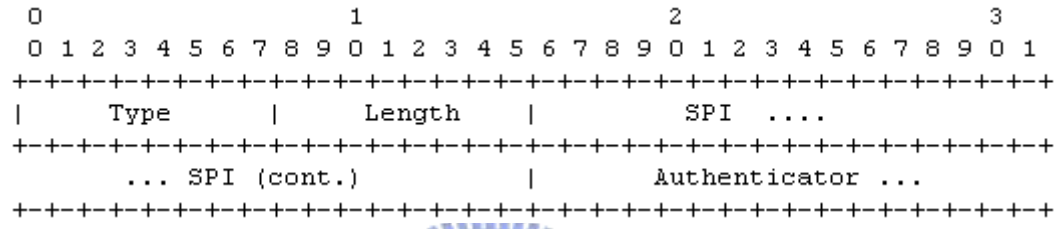

Figure 2 - 9: Authentication Extension Format

由於每個註冊訊息都是由跟隨在固定部分之後的一個或更多的延伸所組成。 驗證延伸必須放置在所有其他延伸之後來保護它們，因為訊息摘要由 UDP 承載所計算出，其中包含了所有之前的延伸。 驗證延伸被放置的順序與可用的安全性關聯有關。 在下面的例子中將會闡明這一點。 圖 2-10 代表一個由行動端經由外地代理器送往家代理器的註冊要求。 在這例子中出現了三個安全性關聯，一個是在行動端與家代理器間(MN-HA, MH)，一個選擇性的在行動端與外地代理器間(MN-FA, MF)以及另一個選擇性的在外地代理器與家代理器間(FA-HA, FH)。 它們每個都有屬於自己的共享金鑰來計算或核對相關的驗證延伸。

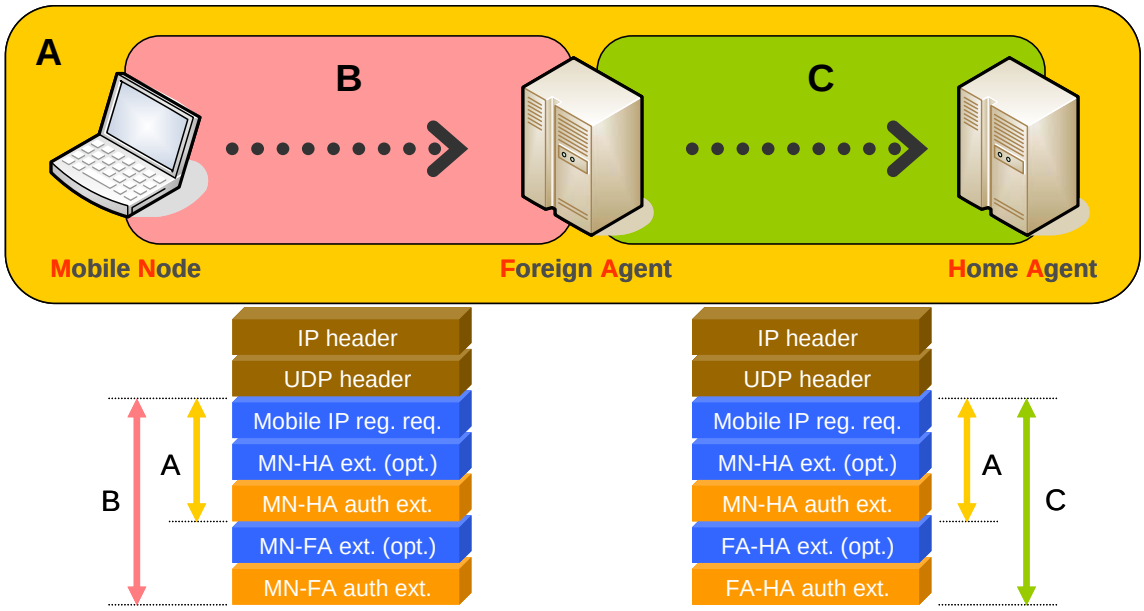

Figure 2 - 10: Authentication Extension Order

當註冊請求封包由行動端被送往外地代理器時，行動端運用箭號 A 所標示的資料計算行動端與家代理器間的驗證元，換言之就是註冊要求，選擇性的 MH 延伸以及前面部分的 MH 驗證延伸(不包含驗證元本身)。 行動端與外地代理器間的驗證元是利用箭號 B 所標示的資料被計算出來。當封包到達外地代理器，外地代理器移除所有的 MF 延伸並核對行動端與外地代理器間的驗證元。 如果正確無誤，它會用箭號 C 所標示的資料計算外地代理器與家代理器間的驗證元。 接著，外地代理器會把它加入封包中並送往家代理器。 家代理器首先會移除所有的 FH 延伸並核對外地代理器與家代理器間的驗證元，之後再移除所有的 MH 延伸並核對行動端與家代理器間的驗證元。 註冊回覆訊息也是以類似的方式被驗證。 在這方法下，安全性關聯是彼此獨立的。 要注意到圖中也展示了一些選擇性未被定義的延伸(MH, MF, FH)用來指出它們被放置的位置。

前面已經討論了註冊訊息以及牽扯到的安全議題。 接下來將深入註冊程序本身。 註冊程序的目的是提供家代理器與選擇性的外地代理器正確的資訊來初始化通道。 對家代理器來說，這些資訊被存在行動連結資訊中。 一個服務數個行動端的外地代理器能對多個家代理器註冊並且對每個註冊保有一個訪客名單(Visitor List)。

當行動端註冊時，它送出一個註冊要求給外地代理器。 在核對選擇性的 MF 驗證延伸過後，外地代理器產生一個待處理的訪客名單登記(Pending Visitor List Entry)。 接著選擇性的加入 FH 驗證延伸並把訊息轉交往家代理器。 待處理的訪客名單登記包含了以下項目：

- 鏈結層來源位址
- IP 來源位址
- IP 目的位址
- UDP 來源埠號

- 家代理器位址
- 識別欄位
- 要求的註冊有效期間
- 剩餘待處理或目前註冊的有效期間

當家代理收到這個註冊要求，它先核對選擇性的 FH 驗證延伸。 然後核對必要的 MH 驗證延伸。 如果這些都是有效的，家代理器便為這個行動端產生一個行動連結資訊，它包含了：

- 行動端的家位址
- 行動端的轉交位址
- 註冊回覆中的識別欄位
- 註冊的剩餘有效期間

家代理器回覆一個註冊回覆送往外地代理器，指出這個註冊是否成功或為什麼失敗。 如果這個註冊是成功的，外地代理器啟用相對應的待處理的訪客名單登記並回覆給行動端。 這個回覆也是被驗證延伸所保護。 現在家代理器便能建立通往轉交位址的通道。 在配置轉交位址模式的情況下，註冊程序直接在行動端與家代理器之間進行，而如果是外地代理器轉交位址模式的情形，則是中間透過外地代理器。 圖 2-11 表現了註冊訊息的交換與通道的建立。

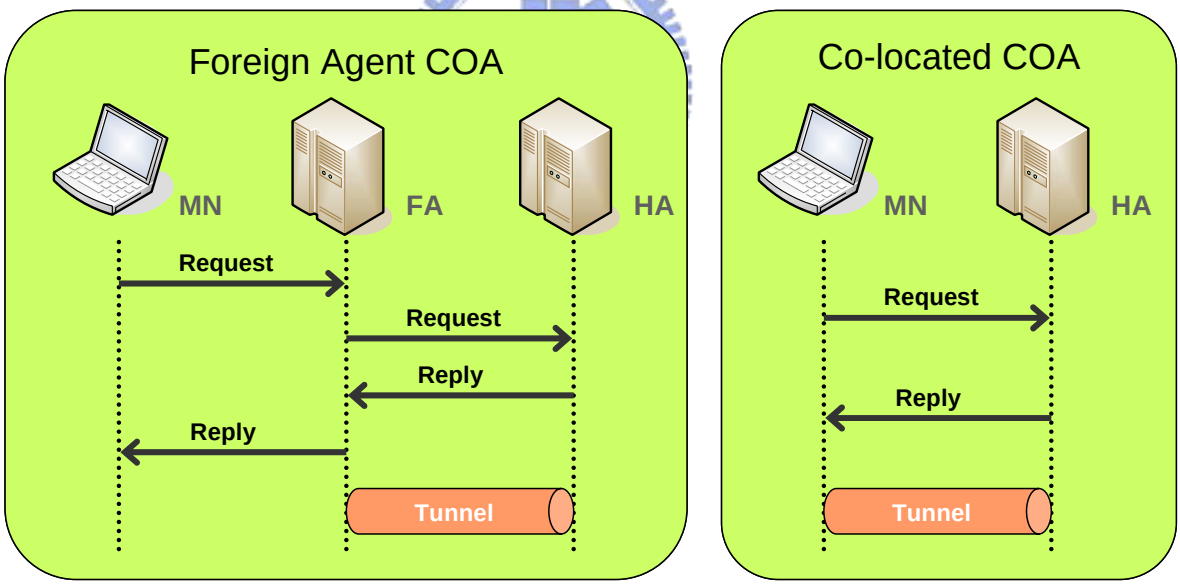

Figure 2 - 11: Registration Flow

家代理器與外地代理器對每個行動端都會保有剩餘的註冊有效期間。 行動端有責任在註冊有效期間到期前重新註冊以維護這個通道。 行動連結資訊在有效期間到期後會被刪除。 當行動端用要求一個註冊有效期間為零的註冊時，行動連結資訊也會被刪除。 所以實際上這就是解註冊 (Deregistration)。 解註冊對於同時擁有數個連結資訊並想刪除其中一個或多個的行動端來說非常有用。 行動端在回到它的家網域後應該解註冊它所有的連結資訊。

2.1.3 Data Encapsulation

在前面的段落中清楚解釋了 Mobile IP 是一種先進的路由機制，實際上就是它使用了動態的通道技術。 Mobile IP 支援了三種通道協定。 基本的通道協定叫做”IP 承載 IP 封裝 (IP within IP Encapsulation)” [9]。 這必須被所有的家代理器，外地代理器以及支援配置轉交位址的行動端所支援。 此外，另外還有兩種通道協定被列入可供選擇： ” 最小化 IP 承載封裝 (Minimal Encapsulation within IP)” [10] 與”通用路由封裝 (Generic Routing Encapsulation, GRE)” [6]。 行動端可以在註冊要求中向家代理器要求其中一種通道模式。 而在外地代理器有參與的情況下，必須外地代理器有公告它支援這兩種額外的封裝行動端才可以要求。

IP 承載 IP 封裝是一種非常基本的通道模式。 IP 封包在一個稱做封裝者(Encapsulator)的端點被封裝，這也就是通道的進入點。 封裝的意思是原本的 IP 封包被裝入一個新的 IP 封包內當作承載，如圖 2-12 所示。 新封包的 IP 標頭稱為傳送標頭，包含了它的拆裝者的(Decapsulator)IP 位址。 該端點會拆裝這個新的 IP 封包來恢復原本的 IP 封包，使得該封包能被路由到它原本的目的地。 在這個通道協定下，拆裝者事實上不過就是把第一個 IP 標頭給移除，同時它也是這個通道的終點。 這種通道技術的目的是把原本的 IP 封包傳送到一個中繼端點，也就是在原本的 IP 標頭中根據 IP 目的位址欄位不會被選擇到的拆裝者。 在 Mobile IP 中， 封裝者是家代理器，而拆裝者則是外地代理器或是行動端。

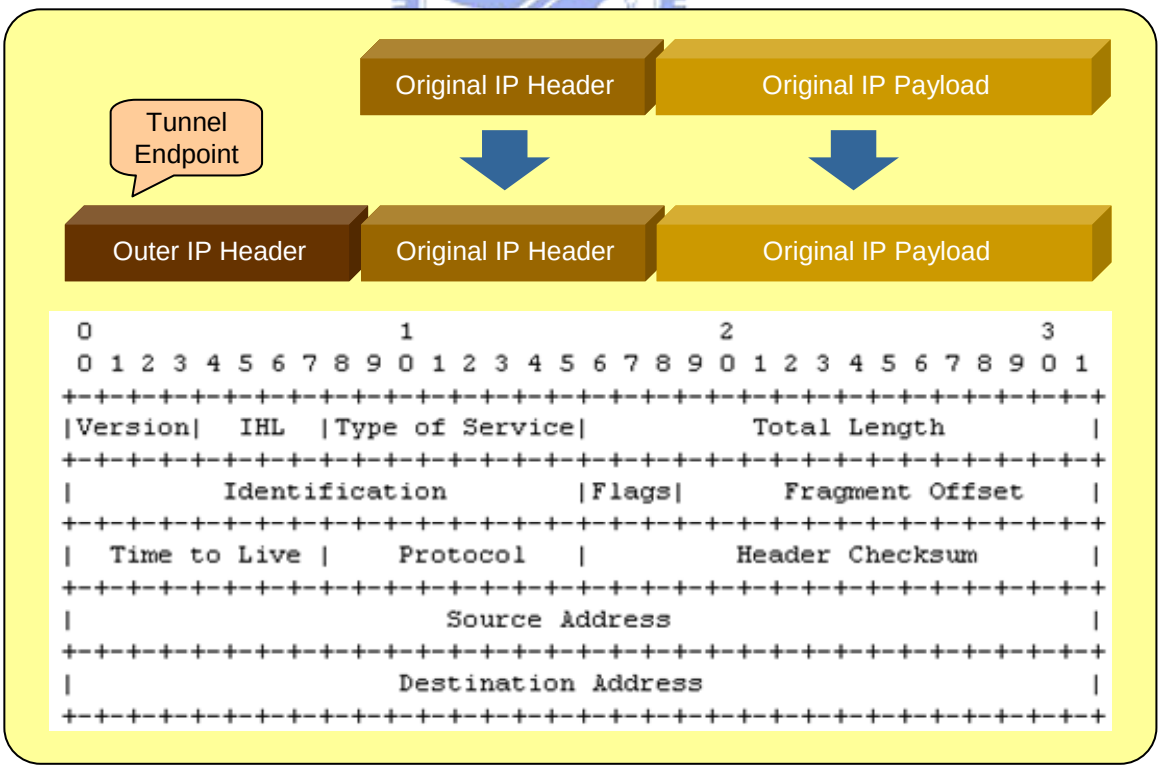

Figure 2 - 12: IP-in-IP Encapsulation and Header Format

最小化 IP 承載封裝是一種非常類似於 IP 承載 IP 封裝的一種通道模式，見圖 2-13。 差別在於最小化 IP 承載封裝不會封裝原本訊息完整的 IP 標頭，因為許多欄位與新的外層傳送標頭是重複的。 在這方式下，標頭空間能被節省，導致更少的負擔。

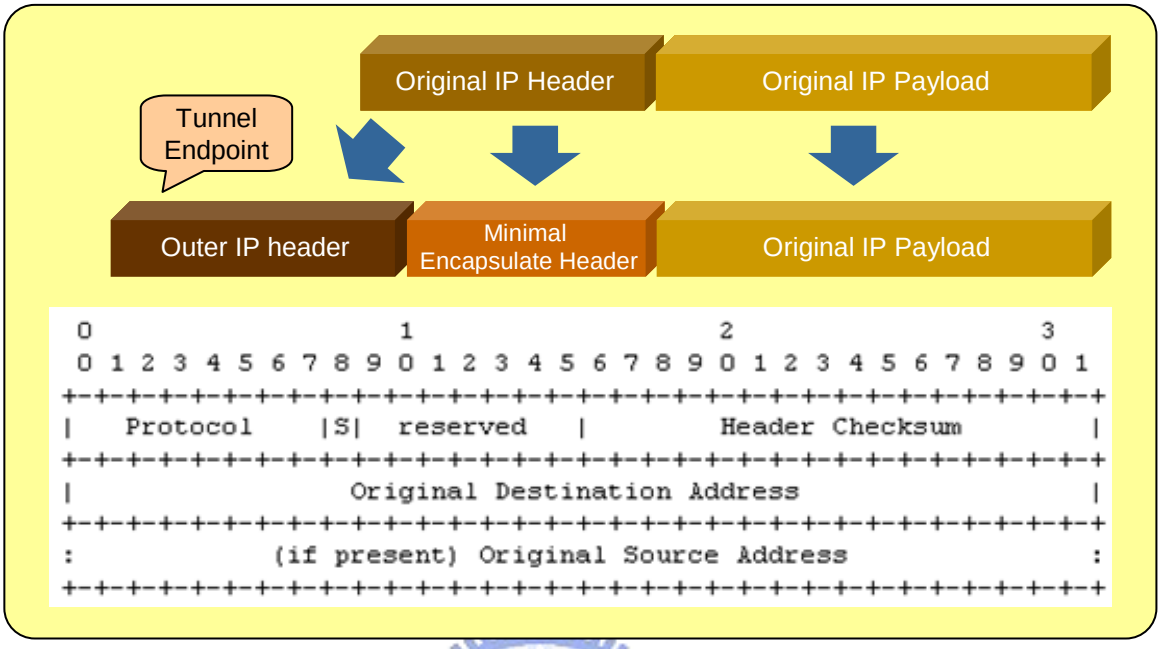

Figure 2 - 13: Minimal Encapsulation and Header Format

通用路由封裝是一個更通用的通道協定，其插入了一個額外的 GRE 標頭在原本的封包與傳送標頭之間，見圖 2-14。 通用路由封裝不是特定用在封裝 IP 封包進另一個 IP 封包。 它可以封裝任何協定 X 的封包進另一個協定 Y 封包的承載中。

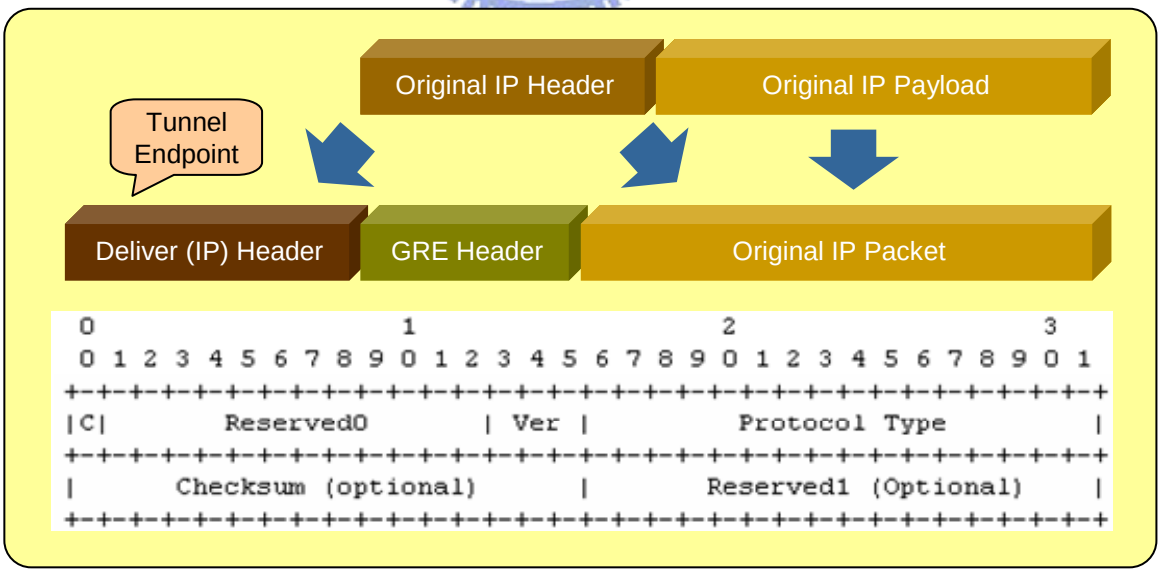

Figure 2 - 14: GRE Encapsulation and Header Format

2.1.4 Specific Problems and Solutions

在這個段落內將討論一些由 Mobile IP 工作群組額外所發表的 IETF 文件，其用途在解決 Mobile IP 運行上遭遇特殊的網路環境與設定時所造成的問題。 以下便仔細探討這些問題的成因與其解決的方法。

- 搭載防假造過濾器的路由器 (Router with Spoofing Filter)

在 2.1.1 中已經提及 Mobile IP 所採用的路由機制可能造成拓樸上錯誤的路由方式。 這會發生在行動端送往通訊端的封包上，當行動端漫遊到外地網域時。 Mobile IP 指出行動端必須在它所送出的 IP 封包中將它的家位址放在來源位址欄位內。 但是這個家位址不屬於外地網域的網路前置碼。 在現今一個自治系統(Autonomous System, AS)的邊界閘道器(路由器)多半有裝配進(Ingress)出(Egress)過濾器。 這些過濾器必須保護網路避免被那些已經被假造 IP 位址的 IP 封包流進或流出這個自治系統。 它的運作方式恰好與路由相反。 當一個 IP 封包通過這個路由器，路由器將使用反向路徑遞送(Reverse Path Forwarding, RPF)來找出這個來源 IP 位址是否合法。 也就是說，如果這個封包所到達的鏈結是其內來源位址所註冊的預設路徑，則此封包將會被遞送。 否則，路由器會捨棄這個封包，因為非常有可能來源位址已經被假造。 這種防假造過濾器造成了行動端所送的封包可能無法到達通訊端。

為了解決這樣的問題，Mobile IP 實作了一個叫做反向通道技術(Reverse Tunneling)的功能，細節可參考 RFC 3024 文件 [13]。 反向通道技術不是必要的。 外地代理器與家代理器會在它們的代理器公告中告知行動端是否支援這樣的功能。 反向通道技術不是在家代理器往轉交位址的方向建立通道，而是由轉交位址往家代理器的方向，圖 2-15 描繪了上述的情形。 因此，如當反向通道技術被使用時，雙向的通道會被建立。

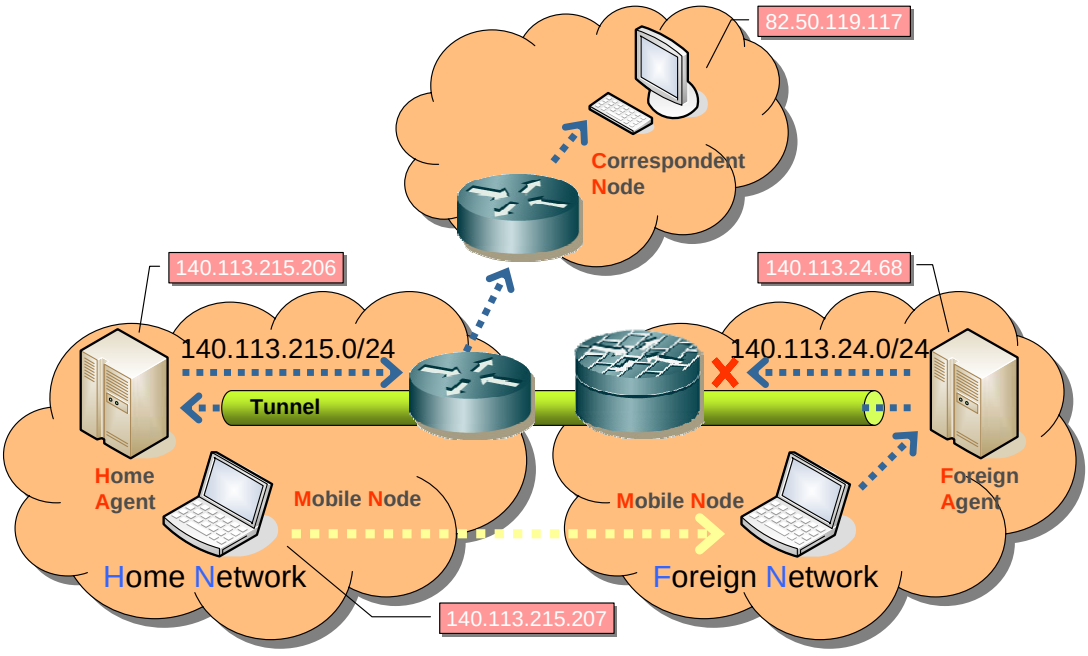

Figure 2 - 15: Reverse Tunneling Routing Scheme

■ 私有網際網路位址環境 (Private IP Address Environment)

為了解決 IP 位址日漸不足的問題，RFC 1918 文件 [7] 中定義了一段私有網際網路位址，這段 IP 位址 可作為企業或單位內自行運用的 IP 位址而無須經過向上游申請的手續。 當然使用單位必須負責不讓這些私有網際網路位址的路由資訊流到單位外的網路上，也就是說使用這些位址的主機只能和單位內的其它主機連線，外面的網路看不見單位內這些私有網際網路位址的主機。 因此這段私有網際網路位址可重覆地被不同單位內部所使用，進而達到節省 IP 位址的目的。 當然這樣的使用方式已不符合現今的需求，因此許多的私有區域網路都透過網路位址轉譯器(Network Address Translator, NAT)來連上網際網路。 關於網路位址轉譯器的詳細運作方式將在下一節介紹。

Mobile IP 做的一個基本假設是行動端與外地代理器皆可被一個全域可路由的 IP 位址所識別。 這個假設當行動端試圖在網路位址轉譯器後端連線時將無法成立。 Mobile IP 靠著藉由 IP 承載 IP 封裝通道將封包從家網域送到行動端或是外地代理器。 在網路位址轉譯器後端連線的行動端只能被網路位址轉譯器的公開位址所找到。 而 IP 承載 IP 封裝因為沒有包含足夠的資訊，所以無法允許從公共的公開位址到處在網路位址轉譯器後端的行動端或是外地代理器轉交位址唯一的轉譯。 特別是當沒有可用的 TCP 或 UDP 埠號供網路位址轉譯器運作。 基於這個原因，IP 承載 IP 通道無法通過一個網路位址轉譯器，也就是 Mobile IP 將無法經由網路位址轉譯器來運作。

而在另一方面，Mobile IP 的註冊要求與註冊回覆由於是 UDP 的資料封包，所以是可以通過網路位址轉譯器的。 當註冊要求通過時，它會使得網路位址轉譯器建立一個位址/埠號的對應，讓註冊回覆得以進到正確的接收端。 所以 Mobile IP 需要的是另一個可供選擇的資料封裝機制提供網路位址轉譯器裝置唯一轉譯的方法以及一個註冊的機制來允許這樣的資料封裝機制在適當的時機被建立。 IETF 組織在 RFC 3519 文件中提出了這樣的解決方案，我們將在稍後的 3.2 節中解釋。

Section 2.2 NAT

在過去的十年內許多 IP 相關的技術引發了某種程度的科技爭論。 其中一項就是網路位址轉譯器(Network Address Translator, NAT)。 本節中將描述網路位址轉譯器內部的運作，首先在 2.2.1 的部分會介紹網路位址轉譯器被開發的原因，而在稍後的 2.2.2 中將深入瞭解其運作的模式。

2.2.1 NAT Motivation

第一份描述網路位址轉譯器的 RFC 文件是由 Kjeld Egevang 與 Paul Francis 於 1994 年所發表的，其編號為 RFC 1631 [4]。 網路位址轉譯器的作用背後原始的動機是基於在 1990 年代初期關於對接替 IPv4 協定的努力。 以整體的貢獻來說，IPv4 的接替協定是設計出一套協定可以直接地解決 IPv4 位址由於加速地消耗所導致即將發生位址消耗殆盡的問題。 雖然 IPv4 有能力定址 44

億個裝置，但早在 1992 年就被證實由於全世界朝向部署具備通訊能力的裝置，而 IPv4 卻無法擴展到未來裝置部署的範圍。 網路位址轉譯器的目的就是制訂一個機制允許 IP 位址被許多裝置所共享。 除此之外，它還預期網路位址轉譯器能以漸進的方式部署在網際網路內而不需造成其它主機或是路由器的改變。 原始的 RFC 描寫這種方法能”當其它更複雜與長期研究的解決方案出現前提供暫時性的替代”。

所以，如同文件所說，最初網路位址轉譯器的意圖是希望當遠程的解決方案正在被設計之時能夠近程的解決位址耗盡的問題。 網路位址轉譯器也希望是未受管理的裝置，對於端對端(End-to-end)的協定是透明的，終端系統與網路位址轉譯器裝置間無須特定的互動。 十年後的現在，網路位址轉譯器達到了幾近普遍存在部署於網際網路上的狀態。 雖然 IPv6 已經被制訂並開始部署，網路位址轉譯器似乎在網路的園地上成為非常根深蒂固的一部分。

2.2.2 NAT Operation

網路位址轉譯器是被放置在資料路徑上的主動單元，通常作為邊界路由器或是區域閘道器的一個作用元件。 網路位址轉譯器攔截所有的 IP 封包並可能修改或未修改封包的內容後向前遞送此封包，或是它也可以決定丟棄這個封包。 與傳統的路由器或是防火牆本質上的差異在於網路位址轉譯器在遞送封包前可以任意修改 IP 封包的能力。 網路位址轉譯器與防火牆相似，但不同於路由器。 它有一個”內部”與一個”外部”，並且對攔截的封包採取不同的運作基於封包是由內部往外部或是相反的方向。

網路位址轉譯器是 IP 標頭的轉譯器，特別是，它是 IP 位址轉譯器。 IP 標頭包含了來源與目的 IP 位址。 如果封包是由內部往外部的方向，網路位址轉譯器將重寫封包標頭中的來源位址。當封包是由外部接收到，目的位址會被重寫。 由 RFC 1631 定義的典型網路位址轉譯器將 IP 位址由一個網域對應到另一個。 雖然它能被用來在任何兩個網域位址間轉譯，但網路位址轉譯器最主要被用在對應定義在 RFC 1918 中不可繞送的私有位址，如下表所示。

Class	Address Range	Address Mask
A	10.0.0.0 ... 10.255.255.255	10.0.0.0/255.0.0.0
B	172.16.0.0 ... 172.31.255.255	172.16.0.0/255.240.0.0
C	192.168.0.0 ... 192.168.255.255	192.168.0.0/255.255.0.0

Table 2 - 1: Private IP Addresses Table

這些位址被分配作為不需要外界存取或是對外界服務需要限制存取的私有網路所使用。 企業可以免費的使用這些位址來避免去獲得已註冊的公開位址。 但是因為私有位址能被許多人，單獨用在其內的網域，因此它們在普通的基礎建設上是不可繞送的。 當擁有私有網路位址的主機需要與公開的網路通訊時，網路位址轉譯器就必須存在。 這也就是網路位址轉譯器的由來。

■ 靜態網路位址轉譯器 (Static NAT)

網路位址轉譯器的運作是藉由在位址間建立連結資訊。 舉個最簡單的例子，公開位址與私有位址間可以定義一個一對一的對應關係。 這就是大家所熟知的靜態網路位址轉譯器，它可以被一個直覺的,無狀態性質的實作所實現。 也就是只轉譯網路前置碼的部分，而不去更動當地位址。在轉譯的過程中也必須考慮封包的承載。 當然，特別是 IP 檢查碼(Checksum)，必須被重新計算。 另外，由於 TCP 的檢查碼是由包含來源與目的 IP 位址的虛擬標頭所計算出，網路位址轉譯器同樣也必須重新產生這個 TCP 的檢查碼。

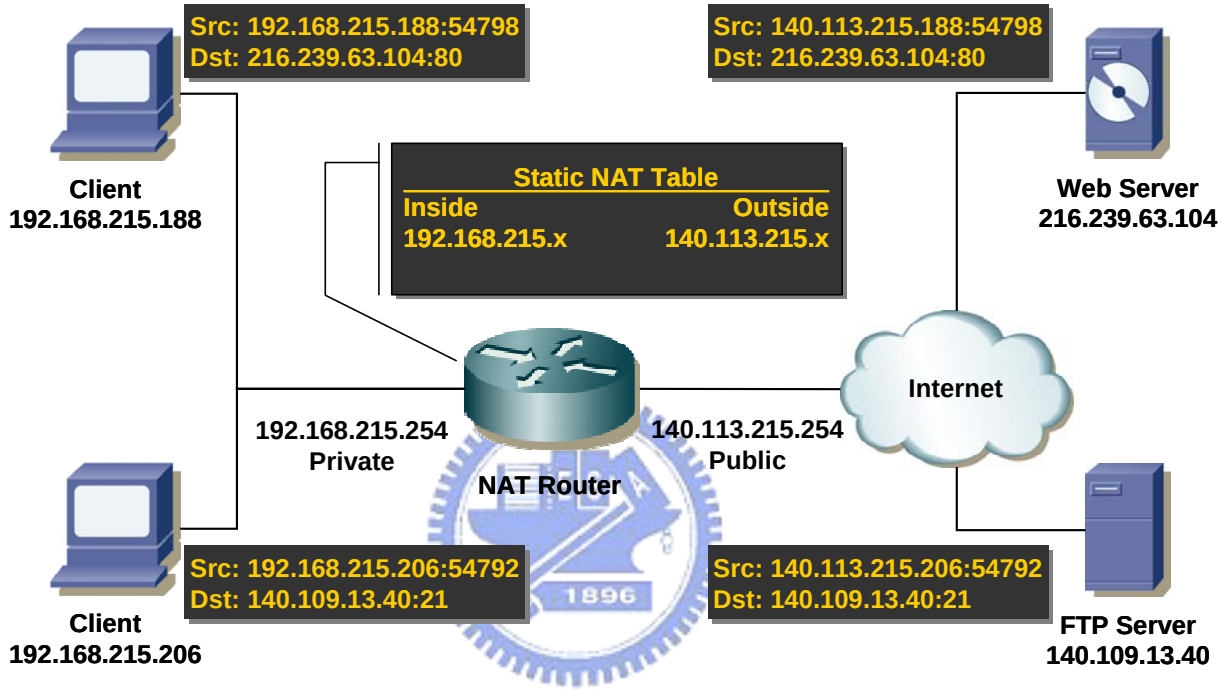

Figure 2 - 16: Static NAT

如圖 2-16 所示，這種形式的網路位址轉譯器在其內維護一個關連內部 IP 位址與對應的外部網際網路 IP 位址的表格。 當使用靜態網路位址轉譯器時，你必須為每一台連上網際網路的機器註冊一個 IP 位址。 這個方式並不常被使用，因為它不能節省公開的 IP 位址。 然而，靜態網路位址轉譯器對於讓裝置被網際網路端存取卻有幫助，由於外部 IP 位址總是指向一個存在網路位址轉譯器上的一個內部 IP 位址。

■ 動態網路位址轉譯器 (Dynamic NAT)

更常見到的是，有一池子(Pool)的公開 IP 位址被整個私有 IP 網域所共享，也就是動態網路位址轉譯器。 它會動態地建立連結資訊，建構出一個 NAT 表格。 由私有主機所初始化的連線會由該池子中被分配一個公開位址。 只要這個私有主機有對外的連線，送往這個公開位址的對內封包就能夠送達它。 當連線被中斷後(或是時效到期)，連結資訊期限終止，這個位址便被回歸到這個池子以供重複使用。 動態網路位址轉譯器是比較複雜的，因為狀態必須被維護，以及當池子耗盡時連線必須被拒絕。

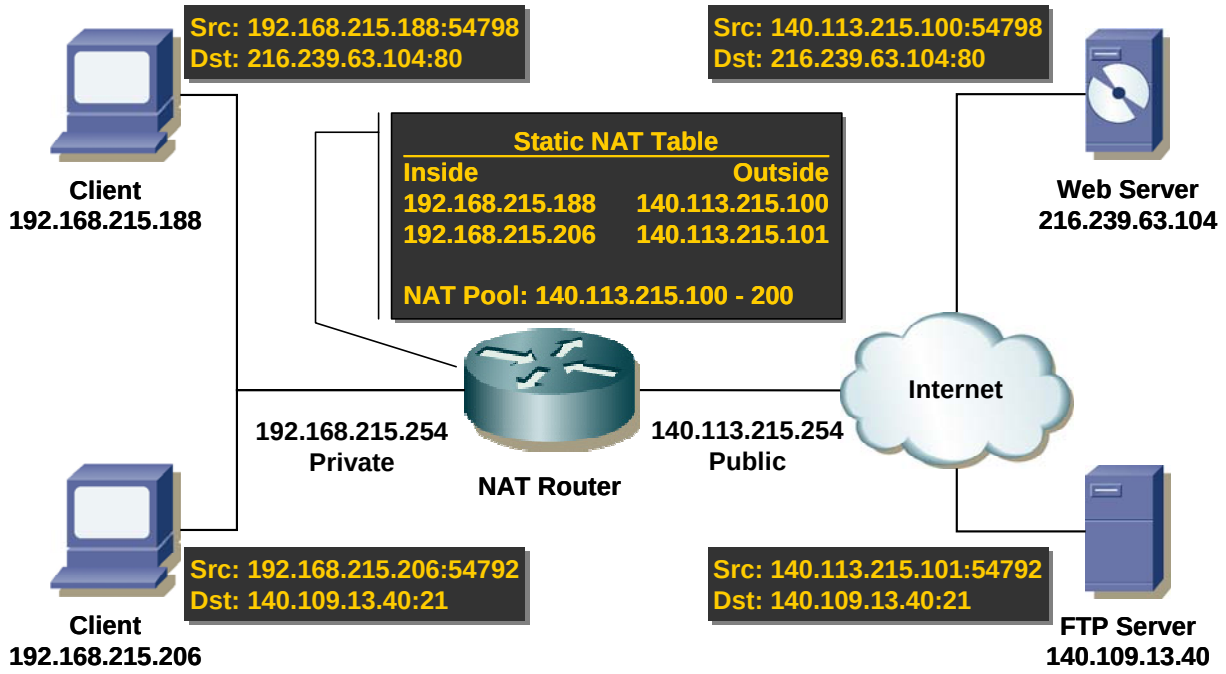

Figure 2 - 17: Dynamic NAT

如圖 2-17 所示，動態網路位址轉譯器維護了一個公開 IP 位址的列表。 每當有內部的客戶端試圖存取網際網路，它便分配一個目前尚未使用的 IP 位址。 因此，你只需要同時間網際網路用戶的數量之公開 IP 位址。 也就是說，動態網路位址轉譯器使得位址可以重複使用，降低了需要合法註冊的公開位址的需求。

- 網路位址埠號轉譯器 (Network Address Port Translation, NAPT)

動態網路位址轉譯器的一種變形叫做網路位址埠號轉譯器，它能被用來讓許多主機透過辨別多路傳輸串流 TCP/UDP 埠號的不同共用一個單獨的 IP 位址。 舉例來說，如圖 2-18 所示，假設私有主機 192.168.215.188 與 192.168.215.206 皆從埠號 54792 送出封包。 網路位址埠號轉譯器會將它們轉譯成同樣的公開 IP 位址 140.113.215.254 與兩個不同的來源埠號，也就是 31892 與 31893。 從埠號 31892 接收到的回應流量將被遞送回 192.168.215.188:54792，而埠號 31893 的流量則被遞送回 192.168.215.206:54792。

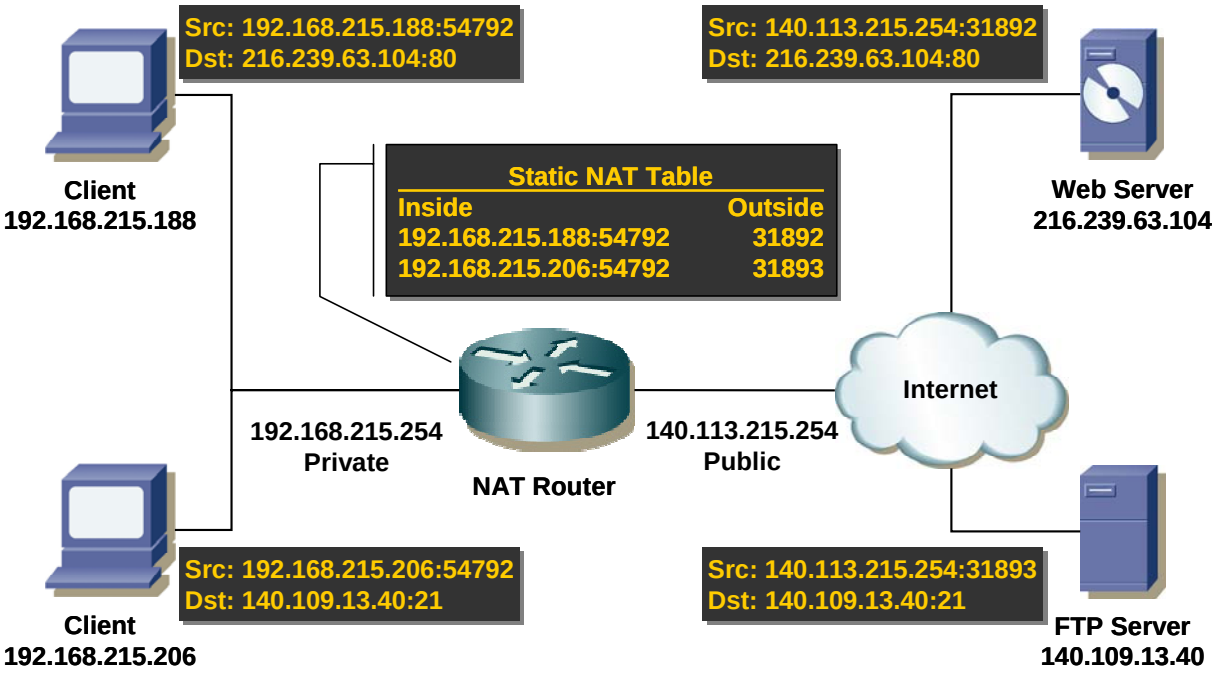

Figure 2 - 18: Network Address Port Translation

網路位址埠號轉譯器又被稱為單一位址網路位址轉譯器(Single-Address NAT)/超載(Overloading)/ 偽裝(Masquerading)，它是小型辦公室和家庭辦公室(Small Office/Home Office，SOHO)中最常被建置的，也是目前市面上最廣泛的一種網路位址轉譯器配置類型。

在爾後的章節中，我們將以網路位址轉譯器泛稱以上的各種配置類型。

Section 2.3 Packet Intercepting Technique on Microsoft Windows

封包的擷取以功能作為區別大致上可分為被動與主動兩方面做探討。 以被動面來說，主要是只聆聽封包的內容而不擅加修改及變更原封包的路由路徑，其主要的應用包括封包解析，流量分析，流量記錄及入侵偵測等。 相反的在主動面，則是積極的介入原封包的內容修改與路由變更，可應用的層面有封包的封裝，防火牆，甚至是網路位址轉譯器或路由器的實作。 此外，在此所指的封包的擷取皆以單一主機所傳送與接收的封包而言。

在本節中將描述在微軟公司的視窗平台上如何達成封包的擷取，內容依序是視窗的網路架構 NDIS 於 2.3.1，用戶模式下的封包擷取方式於 2.3.2，核心模式下的封包擷取方式於 2.3.3，並在最後的 2.3.4 中介紹現有的封包擷取函示庫。

2.3.1 NDIS Overview

Microsoft 與 3Com 公司於 1989 年聯合制訂了一套開發視窗環境下網路驅動程式的規範，稱為 NDIS (Network Driver Interface Specification)。 它使不同的傳輸協定與實體網路卡分離，符合 NDIS 規範的網路驅動程式可直接或稍加修改就可在幾種視窗平台上執行。 這使得視窗軟體開發者可以開發出能夠和多種傳輸協定進行溝通的網路驅動程式。 NDIS 規範了網路驅動程式間的標

準界面，也負責維護網路驅動程式的參數和狀態訊息及其它系統參數。 整體來說，NDIS 本質上是一種軟體程式界面，它使不同的傳輸協定可以採用一種通用的模式來使用由不同廠商所製造的網路卡。

在 NDIS 所訂定下的網路架構是一層級化的組織方式，藉由允許增加額外的功能與服務來提供其擴充性。 網路架構示意圖如圖 2-19 所示，其中包括了兩個重要的層級邊界，TDI 與 NDIS 介面以及三種主要的網路驅動程式，Protocol、Intermediate 與 Miniport 驅動程式。 以下我們將就這些元件做一簡單的介紹。

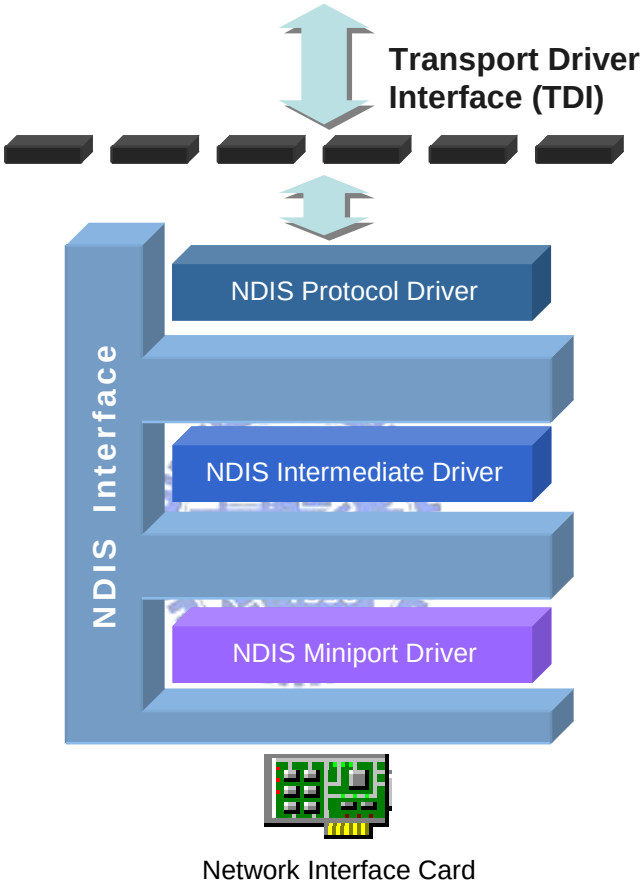

Figure 2 - 19: Network Driver Interface Specification

- 傳輸層驅動程式介面 (Transport Driver Interface, TDI)

傳輸層驅動程式介面定義了一個置於傳輸協定堆疊上緣的核心模式網路介面，如下圖所示。它使得核心模式的驅動程式，也就是所謂的 TDI 客戶端，如 Redirector 與 Server 仍然獨立於傳輸層，只透過 TDI 介面與傳輸層互動。 TDI 簡化了開發傳輸驅動程式的工作，也就是說只剩下 TDI 介面需要被寫作。 而由傳輸驅動程式所提供的 TDI 介面也只能被 TDI 客戶端所使用。 為了增加傳輸層的存取性，在 Windows 2000 以及之後的版本，替現存熱門的網路介面 Windows Sockets 與 NetBIOS 增加了兩個模擬器模組。 各自的模擬器模組提供了其特有的函式庫，能在用戶模式下透

過標準的呼叫所存取。 當被呼叫時，模擬器模組將特有的函式對應到一個或多個 TDI 函式，並透過 TDI 介面呼叫相對應的傳輸驅動程式。

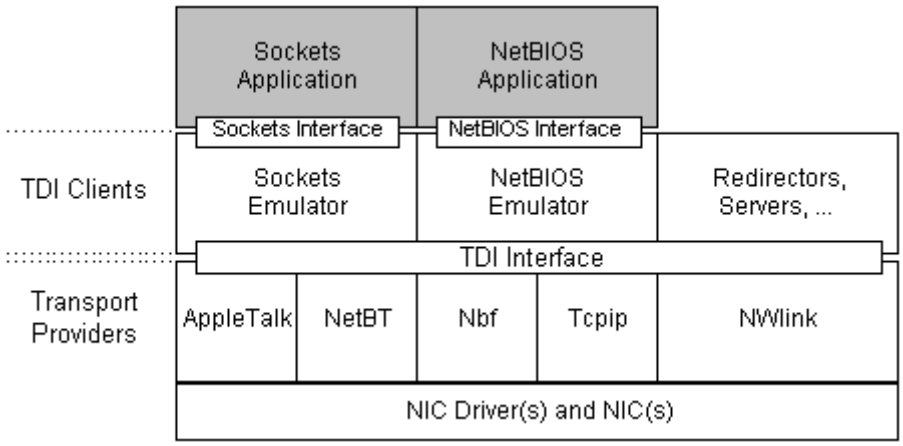

Figure 2 - 20: Transport Driver Interface

- 網路驅動程式介面規範介面 (NDIS Interface)

網路驅動程式介面規範介面也就是所謂的網路驅動程式介面包裝(NDIS Wrapper)，它是包覆在所有 NDIS 裝置驅動程式周圍的小部分程式碼。 這個包裝提供一個單一的介面在各層的驅動程式之間，並包含讓開發 NDIS 驅動程式更簡單的支援函式庫。

- 協定驅動程式 (Protocol Driver)

網路協定驅動程式是在 NDIS 層級中最高的驅動程式，通常在傳輸驅動程式作為最底層的驅動程式，其實作了傳輸協定堆疊，例如我們所熟知的 TCP/IP 或 IPX/SPX。 傳輸協定驅動程式負責配置封包，將應用程式欲傳送的資料放入封包內後，呼叫 NDIS 的函式將這些封包送往底層的驅動程式。 協定驅動程式同時也提供協定介面用來接收由下一層驅動程式所收到的封包。 傳輸協定驅動程式會將收到的資料傳送給適當的應用程式。

- 中間層驅動程式 (Intermediate Driver)

NDIS 中間層驅動程式介面於上層的協定驅動程式與下層的迷你連接埠驅動程式之間。 通常作為以下幾種運用： 轉換不同的網路媒介，例如置於乙太網路(Ethernet)以及記號環網路(Token Ring)傳輸驅動程式與非同步傳輸模式(Asynchronous Transfer Mode, ATM)迷你連接埠驅動程式之間的中間層驅動程式，其功能便是轉換乙太網路以及環狀網路的封包成為非同步傳輸模式的封包，反之亦然。 過濾封包，常見的封包排程器便是其中一個例子。 負載平衡與容錯備援(Load Balancing Fail Over, LBFO)，中間層驅動程式會開出一個虛擬的介面卡給在其上的傳輸驅動程式，但是透過一張以上的實體介面卡發送封包。 最後，中間層驅動程式還可作為監控及收集網路統計資料。

- 迷你連接埠驅動程式 (Miniport Driver)

迷你連接埠驅動程式有兩個基本的功能，其一為管理實體網路卡，包括透過網路卡傳送與接收資料。 其二則為與較高層的驅動程式溝通，例如中間層驅動程式與協定驅動程式。 一般我們安裝網路卡後所安裝的驅動程式便是屬於這一類。

2.3.2 User-mode Approach

在微軟視窗平台用戶模式下的網路架構圖如下所示。 在此模式下封包的擷取方式主要是利用一些系統所提供的動態連結函式庫(Dynamic Link Library, DLL)或是使用者所自行開發的。 以下將介紹可能的實作方式。

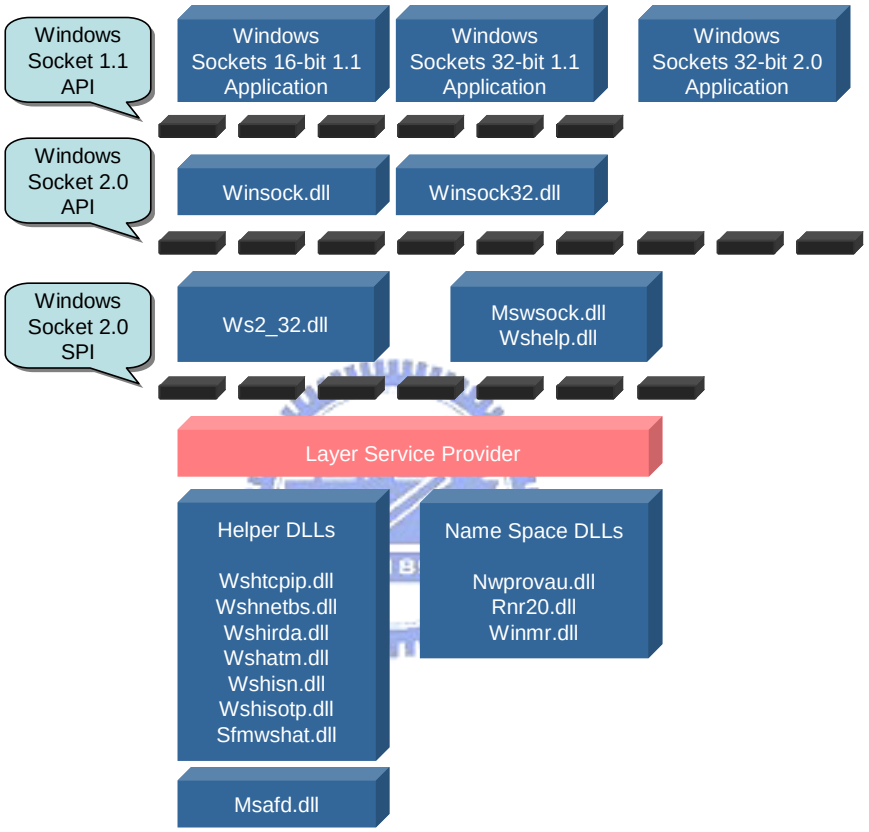

Figure 2 - 21: User-mode Network Architecture Diagram

- Winsock 層級化服務提供者 (Winsock Layered Service Provider, LSP)

根據視窗開放服務架構(Windows Open Services Architecture, WOSA)，應用程式介面(Application Programming Interface, API)提供應用程式開發者存取網路的服務，而服務提供者介面(Service Provider Interface, SPI)則由傳輸服務提供者，名稱解析服務提供者以及 ws2_32.dll 所實作。 由於 SPI 工作在 API 之下，ws2_32.dll 裡的 API 被應用程式呼叫後大部分最終對應到 SPI 裡的函式。 利用這點便可以擷取所有基於 Winsock 的傳輸。 因此，我們可以基礎協定上，例如 TCP 或 SPX 上實作支援 SPI 的層級化協定來達到我們所要的功能，如圖 2-22 所示。 但是運用此方法必須注意到，由於所有的運作都是基於 Winsock，對於那些不經過此的封包，例如 ICMP 等協定，便無法透過這個方法擷取。

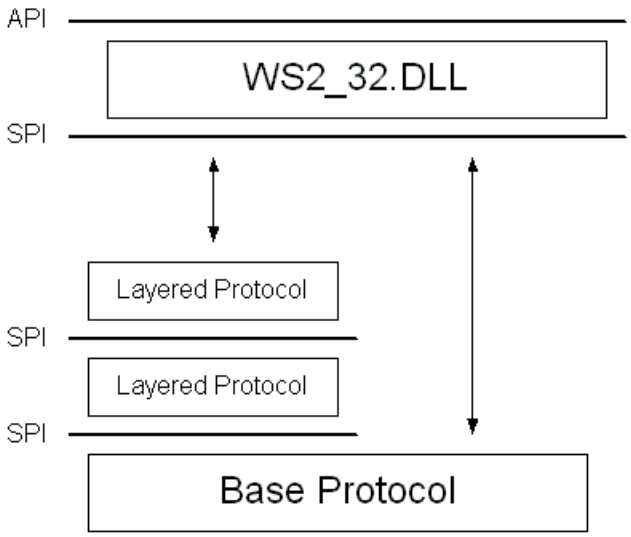

Figure 2 - 22: Transport Service Provider

- Windows 2000 封包過濾介面 (Windows 2000 Packet Filter Interface)

在微軟 Windows 2000 以及之後版本的視窗作業系統，其內的 IPHLPAPI API 提供了較好對 TCP/IP 的可程式化控制，其中包括封包過濾的能力。 但不幸的是，此方法只能針對 IP 位址與 TCP/UDP 埠號進行過濾，且無取得封包內容。 以下是一個阻擋 HTTP 服務的例子。

```
#include "windows.h"
#include "Fltdefs.h"

int main(int argc, char* argv[]) {
  INTERFACE_HANDLE hInterface;
  PfCreateInterface(0,
  PF_ACTION_DROP, //default action for an input packet: drop
  PF_ACTION_DROP, //default action for an output packet: drop
  FALSE,
  TRUE,
  &hInterface);

  BYTE localIp[] = {140.113.215.188}; //local IP address
  PfBindInterfaceToIPAddress(hInterface, PF_IPV4, localIp);

  FILTER_HANDLE fHandle;

  PF_FILTER_DESCRIPTOR inFilter;
  inFilter.dwFilterFlags = FD_FLAGS_NOSYN;
  inFilter.dwRule = 0;
  inFilter.pfatType = PF_IPV4;
  inFilter.SrcAddr = localIp;
  inFilter.SrcMask = "\xff\xff\xff\xff";
  inFilter.wSrcPort = FILTER_TCPUDP_PORT_ANY;
  inFilter.wSrcPortHighRange = FILTER_TCPUDP_PORT_ANY;
  inFilter.DstAddr = 0;
  inFilter.DstMask = 0;
  inFilter.wDstPort = 80;
  inFilter.wDstPortHighRange = 80;
  inFilter.dwProtocol = FILTER_PROTO_TCP;
  // Add Filter
  PfAddFiltersToInterface(hInterface, 1, &inFilter, 0, NULL, &fHandle);
  Sleep(60000); // Block HTTP Service for 60 seconds
  // Remove Filter
  PfRemoveFilterHandles(hInterface, 1, &fHandle);
  PfUnBindInterface(hInterface);
  PfDeleteInterface(hInterface);
  return 0;
}
```

Figure 2 - 23: Windows 2000 Packet Filter Interface Example

■ 替換 Winsock 動態連結函式庫 (Winsock Replacement DLL)

假如我們經過適當的實作，經由換掉微軟所提供的 Winsock 動態連結函式庫後，替代的動態連結函式庫可以過濾 Winsock API 並呼叫原來的動態連結函式庫。 但是有很多原因造成開發一個健全的替代動態連結函式庫很難達成，其中最主要的原因在於微軟所提供的 Winsock 動態連結函式庫包含了很多只提供內部使用的未公開函式。 而替代動態連結函式庫至少必須處理其中一些未公開的函式，使得這方法變的不太可行。 除此之外，使用替代的動態連結函式庫是個不錯的主意，但是如同前述，不經過 Winsock 的封包同樣無法用此方法擷取。

2.3.3 Kernel-mode Approach

在微軟視窗平台核心模式下的網路架構圖如下所示。 在此模式下封包的擷取方式主要是利用系統預先在網路核心內埋入的核心驅動程式或是由使用者自行開發後插入網路堆疊內的。 以下將介紹可能的實作方式。

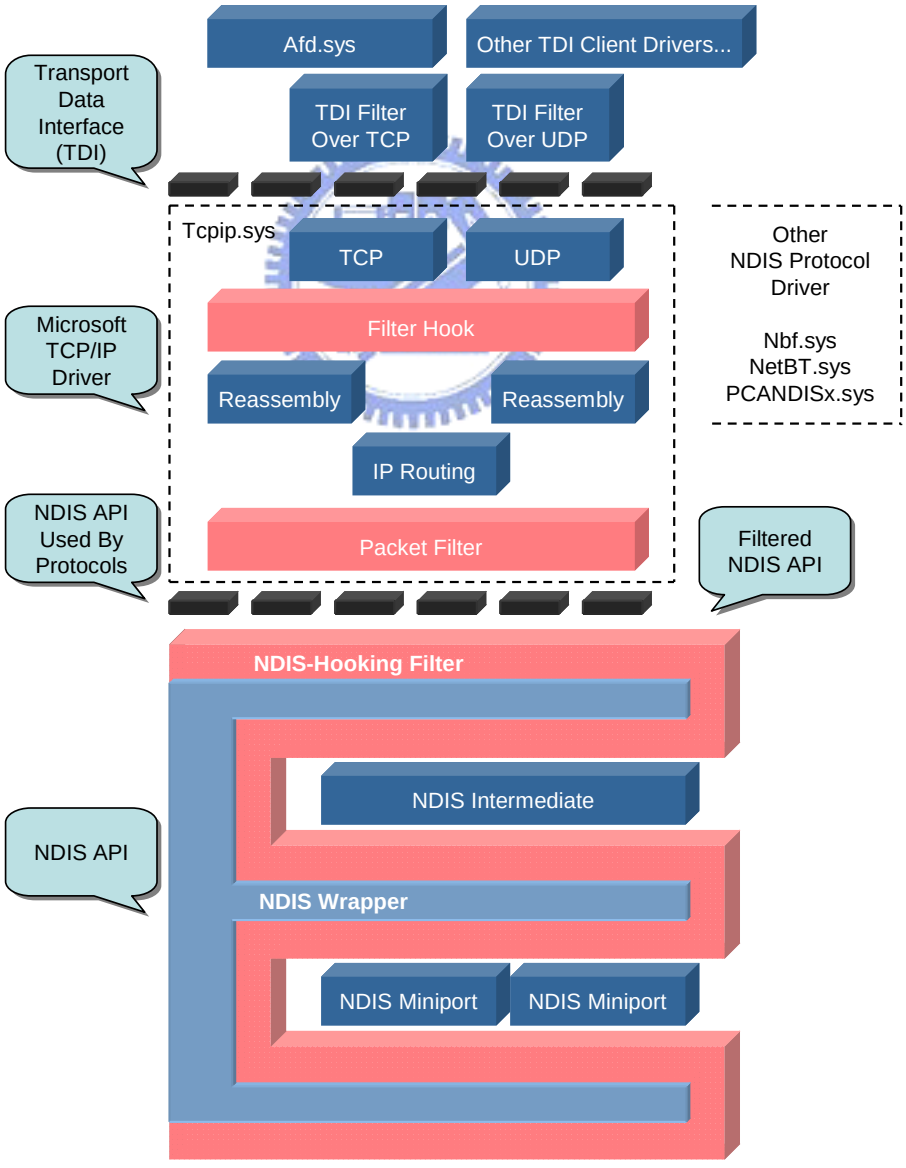

Figure 2 - 24: Kernel-mode Network Architecture Diagram

- 傳輸層驅動程式介面過濾驅動程式 (Transport Driver Interface Filter Driver)

當應用程式要傳送或接收網路封包的時候，都是透過與協定驅動程式所提供的介面，也就是所謂的傳輸層驅動程式介面來進行的。 協定驅動程式提供了一套系統預先定義的標準界面來和應用程式之間進行互動。 在微軟視窗作業系統下 IP，TCP 與 UDP 等協定是在一個驅動程式內實作的，也就是上圖中的 Tcpip.sys。 這個驅動程式建立了幾個裝置： \Device\RawIp、\Device\Udp、\Device\Tcp、\Device\Ip 與\Device\MULTICAST。 應用程式所有的網路封包運作都是透過這幾個裝置來進行。 因此，我們只需要開發一個過濾驅動程式擷取這些互動的介面，便可以實現網路封包的擷取。 然而，由於 TDI 運作在 Tcpip.sys 之上，這就意味著由 Tcpip.sys 接收並直接處理的封包就不會傳送到該介面，因此便無法擷取這類型的封包。 典型的範例就是 ICMP 回應請求(Echo Request)與回應回覆(Echo Reply)，這些封包是由 Tcpip.sys 接收與傳送，在其上的過濾驅動程式將無法得知此事件的發生。

- NDIS 中間層驅動程式 (NDIS Intermediate Driver)

中間層驅動程式介於協定驅動程式與迷你連接埠驅動程式之間，它能夠擷取所有且完整的網路封包。 舉例來說，如果底層是乙太網路迷你連接埠驅動程式，那麼由中間層驅動程式所擷取的資料便是包含來源與目的 MAC 位址的完整乙太訊框(Ethernet Frame)。 運用此種方式開發擷取應用程式的缺點在於安裝驅動程式上，雖然可以透過程是進行自動安裝，但是如果驅動程式沒有經過數位簽章的話，系統會提示使用者是否繼續安裝，造成困擾。 對於非商業化產品的程式開發者來說，取得驅動程式的數位簽章是件繁瑣且無實質作用的一件事情。 除此之外，中間層驅動程式功能強大，是實作擷取網路封包的其中一個不錯的選擇。

- Windows 2000 過濾攔截驅動程式 (Windows 2000 Filter-Hook Driver)

過濾攔截驅動程式是從微軟 Windows 2000 版本的視窗作業系統開始支援的一個核心驅動程式，該驅動程式主要是利用 ipfltdrv.sys 所提供的功能來擷取網路封包。 其架構非常簡單，容易實作。 主要的動作是向 ipfltdrv.sys 註冊一個稱為過濾攔截的回呼函式(Callback Function)，便可以擷取並過濾 IP 封包。 圖 2-25 為用來阻擋所有 TCP 連線的回呼函式範例。 由於其架構過於簡單，因此在使用上便有許多的限制。 首先，系統內只允許一個過濾攔截驅動程式的存在，這對於實際的應用上造成相當大的限制。 此外，雖然過濾攔截回呼函式可以擷取將被傳送與待接收的封包，但是並不保證是完整的封包內容，在一般的情況下，往往只能獲得封包的 IP、TCP 或 UDP 等的標頭部分，這將限制應用的範圍。 最嚴重也最需被注意的是，過濾攔截驅動程式被認定是系統內可被信任的元件，換句話說，只要在其內執行錯誤的指令或是違反記憶體的存取規則都有可能造成整個系統的錯誤。 基於以上種種的缺點，微軟並不推薦採用這項方法。

```
PF_FORWARD_ACTION
DropTcpPackets(
      unsigned char   *PacketHeader,
      unsigned char   *Packet,
      unsigned int    PacketLength,
      unsigned int    RecvInterfaceIndex,
      unsigned int    SendInterfaceIndex,
      IPAddr          RecvLinkNextHop,
      IPAddr          SendLinkNextHop
      )
{
   if (PacketHeader->iph_protocol == PROT_TCP)
   {
      return PF_DROP;
   }
   return PF_FORWARD;
}
```

Figure 2 - 25: Windows 2000 Filter-Hook Driver Example

- NDIS 攔截驅動程式 (NDIS-Hooking Driver)

NDIS 攔截驅動程式是適用範圍較廣泛的一種核心模式封包擷取驅動程式，自 Windows 95 開始到現今的 Windows 2000/XP/2003 皆可使用，且克服了上述方法中的一些缺陷。 簡單的來說，其運作原理就是直接置換由 NDIS 介面包裝所提供函式庫的函式位址。 如此一來，所有呼叫 NDIS 的請求，將會先經過我們自己的函式來處理，最後再轉送給原系統的函式就完成了。 至於如何置換這些函式位址有以下兩種方式：

- 修改 ndis.sys 的匯出表(Export Table)

  在視窗作業系統下，可執行程式(包含 DLL 及 SYS)都必須遵從可移植執行檔格式(Portable Executable File Format)。 所有向其他作業系統元件提供介面的驅動程式都有匯出表，因此只要修改 ndis.sys 的匯出表就可以達成置換 NDIS 的 API，而主要的函式有 NdisRegisterProtocol() 、 NdisDeRegisterProtocol() 、 NdisOpenAdapter() 與 NdisCloseAdapter()。 再置換上述 API 之後，置換呼叫 NdisOpenAdapter()所得到的 NDIS_OPEN_BLOCK 資料結構指標，該資料結構定義在 ndis.h 檔案內。 它的重要性在於其內記錄了傳送與接收封包的函式指標 SendHandler、SendPacketHandler 與 TransferDataHandler。 如此一來便達成了擷取封包的動作。 使用這種方式還必須注意到驅動程式的掛載順序，攔截驅動程式必須在 ndis.sys 之後被掛載，而其他協定驅動程式如 tcpip.sys 則需在它之後。 除此之外，這種方式是屬於靜態的置換，換句話說，作業系統啟動後無法從記憶體中將它卸載。

- 註冊空殼協定(Dummy Protocol)

  在視窗核心內，所有已註冊的協定是透過一個單向的協定鏈結表來維護的。 這個單向鍊結表保存了所有已註冊協定的 NDIS_PROTOCOL_BLOCK 資料結構，該資料結構

是由 NDIS_PROTOCOL_CHARACTERISTICS、NDIS_OPEN_BLOCK 與一個指向下一項 NDIS_PROTOCOL_BLOCK 資料結構的指標所組成。 但不幸的是，該資料結構在不同版本的 NDIS 有不同的定義，而且在 ndis.h 檔案中不一定存在它的定義。 當我們所註冊的空殼協定驅動程式呼叫 NdisRegisterProtocol 之後，NDIS 會把新註冊的協定放在鍊結表的開頭並傳回鍊結表的指標，所以只要我們便可以瀏覽整個鏈結表並置換我們所需要的函式指標。 由上可知，這個方法是屬於動態的置換，可在需要的時候加入或移除，不過此方式對平台的依賴性較大，需要在程式中判斷不同的作業系統版本而使用不同的資料結構定義。

整體來說，NDIS 攔截驅動程式雖然以實作的方法來說有些不正規，但是由於其功能強大，可以擷取所有使用 NDIS 與 TDI 介面傳送的封包，且擷取層級低，應用程式甚至是驅動程式都難以繞過這樣的攔截方法。 設計良好的 NDIS 攔截驅動程式對於網路封包的過濾會非常的有效。 除此之外它的好處還包括安裝容易與支援撥號點對點配接卡(Dialup-PPP Adapter)等。

Section 2.4 Routing Mechanism under Microsoft Windows

當主機名稱或 NetBIOS 名稱被解譯成 IP 位址之後，這個 IP 封包必須被發送主機傳送到解譯出的 IP 位址。 路由便是基於目的 IP 位址繞送封包的過程，其發生在發送的 TCP/IP 主機與 IP 路由器之間。 而無論是發送主機或是路由器，都必須對於封包該往何處繞送做出一個決定。 為了做出這樣的決定，IP 協定查閱存在記憶體中的路由表。 路由表內的預設欄位在 TCP/IP 初始化時就被建立，而其他欄位則由系統管理者手動加入或透過與其他路由器通訊後自動加入。 在本節中將一一介紹微軟視窗環境下的路由機制，分別是 2.4.1 的直接與間接遞送，2.4.2 的路由表與 2.4.3 的實體位址解譯。

2.4.1 Direct and Indirect Delivery

被繞送的 IP 封包基於這個封包是否被繞送到最終目的或繞送到一個路由器來使用兩種遞送型態中的一種。 這兩種遞送型態即為大家所熟知的直接遞送與間接遞送。

- 直接遞送 (Direct Delivery)

直接遞送發生在 IP 端點(無論是發送端或 IP 路由器) 在直接連接的網路上繞送封包到最終目的。 IP 端點封裝 IP 數據資料成為一個送往目的實體位址的網路介面卡層形式訊框(例如乙太網路或記號環網路)。

- 間接遞送 (Indirect Delivery)

間接遞送發生在 IP 端點(無論是發送端或 IP 路由器)繞送封包到一個中間端點(IP 路由器)，由於最終目的不是在直接連接的網路上。 IP 端點封裝 IP 數據資料成為一個送往 IP 路由器實體位址的網路介面卡層形式訊框(例如乙太網路或記號環網路)。

IP 路由即是直接遞送與間接遞送的組合。 在圖 2-16 中，當傳送封包給 B 端點，A 端點採用直接遞送。 當傳送封包給 C 端點，A 端點則採用間接遞送給路由器 1。 路由器 1 也採用間接遞送給路由器 2，路由器 2 則採用直接遞送給 C 端點。

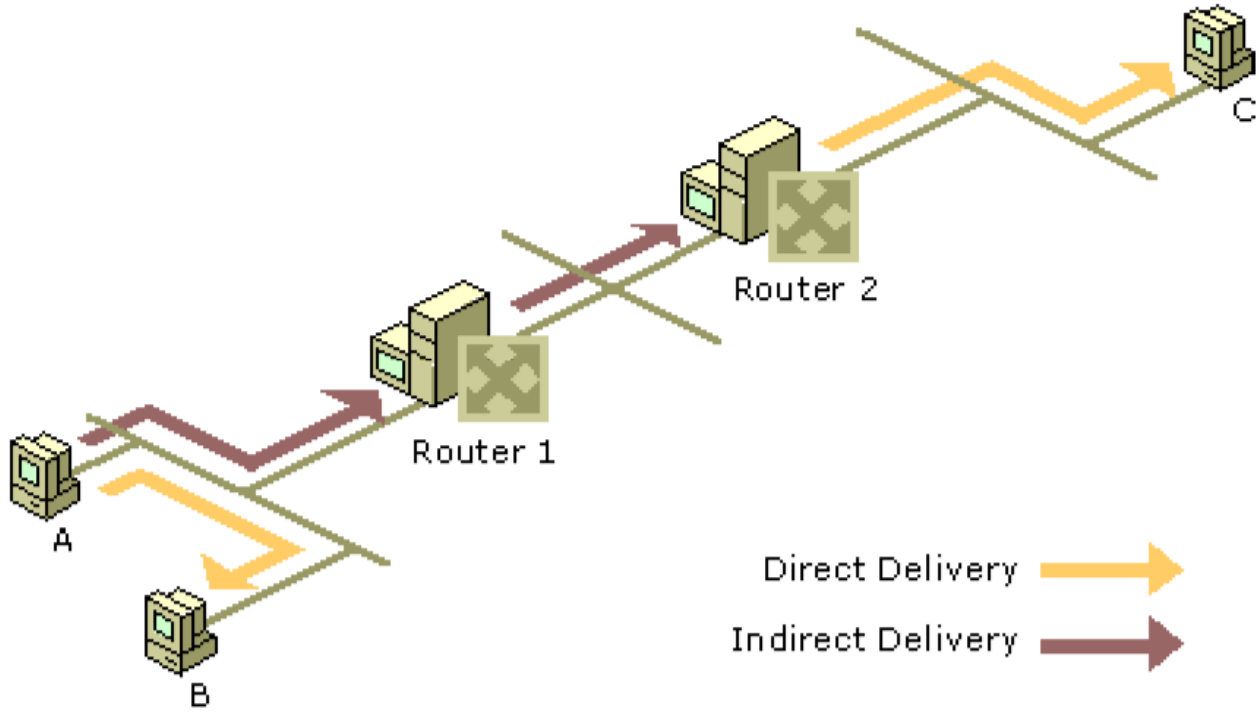

Figure 2 - 26: Direct and Indirect Deliveries

2.4.2 Routing Table

路由表儲存了關於 IP 網路的資訊以及如何到達它們(無論直接或是間接)。 因為所有的 IP 端點都執行某種形式的 IP 繞送，所以路由表並非是 IP 路由器專有的。 任何有載入 TCP/IP 協定的端點皆有一個路由表。 其內有一系列根據該端點設定的預設欄位與能經由 TCP/IP 公用程式手動加入或透過動態地與路由器互動所新增的額外欄位。 當一個 IP 封包將被繞送時，路由表被用來決定兩件事：

- 繞送或次躍點(Next-hop)的 IP 位址

  對直接遞送來說，繞送 IP 位址即是 IP 封包內的目的 IP 位址。 而對於間接遞送，繞送 IP 位址則是路由器的 IP 位址。

- 被用來繞送的介面

  介面確認了被用來繞送封包到目的或下個路由器的實體或邏輯介面。

■ 路由表欄位型態

一個路由表的欄位包含了以下資訊：

- 網路識別碼 (Network ID)

路由所對應的網路識別碼或目的。 網路識別碼可以是以類別為基礎的(class-based)，子網域(subnet)，巨網路識別碼(supernet network ID)或是一個主機的 IP 位址。

- 網路遮罩 (Network Mask)

  遮罩是用來比對目的 IP 位址與網路識別碼。

- 次躍點 (Next Hop)

  次躍點的 IP 位址。

- 介面 (Interface)

  使用何張網路介面繞送 IP 封包的指示。

- 公制 (Metric)

  用來指出這個路由代價的數字，當有多個可能到達相同目的的路由存在時便可在其中選出最佳的一個。 普遍的用法是指出到達該網路識別碼所需要的跳躍數(所經過的路由器數目)。

Network Destination	Netmask	Gateway	Interface	Metric	Purpose
0.0.0.0	0.0.0.0	157.55.16.1	157.55.27.90	1	Default Route
127.0.0.0	255.0.0.0	127.0.0.1	127.0.0.1	1	Loopback Network
157.55.16.0	255.255.240.0	157.55.27.90	157.55.27.90	1	Directly Attached Network
157.55.27.90	255.255.255.255	127.0.0.1	127.0.0.1	1	Local Host
157.55.255.255	255.255.255.255	157.55.27.90	157.55.27.90	1	Network Broadcast
224.0.0.0	224.0.0.0	157.55.27.90	157.55.27.90	1	Multicast Address
255.255.255.255	255.255.255.255	157.55.27.90	157.55.27.90	1	Limited Broadcast

Table 2 - 2: Windows-based Host Routing Table

表 2-2 展示了視窗環境主機(非路由器)的預設路由表。 這個主機只有一張網路介面卡並擁有 IP 位址 157.55.27.90、網路遮罩 255.255.240.0(/20)與預設閘道 157.55.16.1。路由表欄位能用來存放以下型態的路由：

- 預設路由 (Default Route)

  這個對應到預設閘道設訂的欄位是一個網路目的為 0.0.0.0 以及網路遮罩為 0.0.0.0。因此，對於任何 IP 位址來說，預設路由皆能產生匹配(match)。 如果預設路由因為沒有其他更好的路由存在而被選擇，該 IP 封包會用介面(Interface)欄內 IP 位址所對應到的介面被繞送到閘道(Gateway)欄內的 IP 位址

- 回送網路 (Loopback Network)

  回送網路欄位是被設計用來處理形式為 127.x.y.z 的任何 IP 位址並將之繞送到一個特殊的回送位址 127.0.0.1。

- 直接連接的網路 (Directly Attached Network)

  本地網路欄位對應到直接連接的網路。 送往直接連接的網路之 IP 封包不會被繞送到路由器而是直接地傳送到目的。 要注意到閘道欄與介面欄對應到端點的 IP 位址。 這意味該封包是從該端點 IP 位址所對應到的網路介面卡送出。

- 本地主機 (Local Host)

  本地主機欄位是一個對應到該主機 IP 位址的主機路由(網路遮罩為 255.255.255.255)。 所有往該主機 IP 位址的 IP 數據資料會被繞送到回送位址.

- 網路廣播 (Network Broadcast)

  網路廣播欄位是一個對應到所有子網域直接廣播位址(B 類別的所有子網域網路識別碼為 157.55.0.0)的主機路由(網路遮罩為 255.255.255.255)。 往所有子網域直接廣播位址的封包是從對應到該端點 IP 位址的網路介面卡所送出。

- 多點傳送位址 (Multicast Address)

  擁有 D 類別網路遮罩的多點傳送位址是被用來繞送任何來自該端點 IP 位址所對應到的網路介面卡之多點傳送 IP 封包。

- 被限制的廣播 (Limited Broadcast)

  被限制的廣播位址是一個主機路由(網路遮罩為 255.255.255.255)。 被限制的廣播之封包是由該端點 IP 位址所對應到的網路介面卡所送出。

■ 路由決策流程

為了決定該使用那一個路由表欄位作為繞送決策，IP 協定使用了以下的流程：

- 對於路由表內的每一項欄位，執行目的 IP 位址與網路遮罩的位元單位(bit-wise)邏輯與(AND)運算。 將其結果與該欄位的網路識別碼做比對尋求匹配。
- 符合的路由清單會被彙整。 擁有最長匹配(longest match)的路由(與目的 IP 位址最多位元符合的路由)將被選擇。 最長匹配的路由是往目的 IP 位址最明確的路由。
- 如果有多個擁有最長匹配的欄位被找到(舉例來說，多個往相同網路識別碼的路由)，則使用最低公制來選擇最佳的路由。
- 假設存在多個最長匹配且具有相同公制的欄位，則該主機將隨機選擇使用其中一個路由表欄位。

當由選定的路由決定繞送或次躍點的 IP 位址，IP 協定使用了以下的流程：

- 如果閘道位址與介面位址相同，繞送 IP 位址被設定為 IP 封包內的目的 IP 位址。
- 如果閘道位址與介面位址不同，繞送 IP 位址被設定為閘道位址。

路由決策過程的結果是在路由表內選擇一個單一的路由。 被選擇的路由則產生出一個繞送 IP 位址(也就是次躍點 IP 位址)與一張介面卡。 如果路由決策過程無法找到一個路由，IP 協定會宣告路由錯誤。 對於發送主機來說，IP 路由錯誤將在系統內部知會上層協定，如 TCP 或 UDP。

■ 多重網路連線

當一台電腦設定有一個以上的 IP 位址，其被稱為一個多重網路連線的(multihomed)系統。 多重網路連線由以下三種方式被支援：

- 單一網路卡多個 IP 位址
- 單一實體網路多張網路卡
- 多重網路與媒介型態

當 IP 數據資料由一個多重網路連線的主機所送出，IP 路由決策機制會決定適當的繞送 IP 位址與介面。 因此，包含多重網路連線主機其中一張介面 IP 位址的數據資料可能會放置到另一張介面的媒介上。 在訊框內的來源 MAC 位址為真正在媒介上傳送該訊框的介面，而來源 IP 位址則是來自發送的應用程式，不一定要是傳送介面對應的 IP 位址。

當多重網路連線電腦以網路卡連接在不同的網路區段，便有額外的路由考量。 如果有可能為每一個網路介面設定其預設閘道 IP 位址，則在路由表內只會有一個單一有效的預設路由。 假設在路由表內存在多個預設路由，明確的預設路由會在 TCP/IP 初始化時隨機地被選擇。

2.4.3 Physical Address Resolution

基於目的 IP 位址與路由決策過程，IP 協定決定了被用來繞送該封包的繞送 IP 位址與介面。接著 IP 協定將 IP 封包，遞送 IP 位址與介面交給 ARP 協定：

- 如果遞送 IP 位址與目的 IP 位址相同，ARP 協定執行直接遞送。 在直接遞送上，目的 IP 位址對應到的 MAC 位址必須被解譯。
- 如果遞送 IP 位址與目的 IP 位址不同，ARP 協定執行間接遞送。 此時遞送 IP 位址為目前 IP 端點與最終目的之間的路由器 IP 位址。 在間接遞送上，路由器 IP 位址所對應到的 MAC 位址必須被解譯。

為了解譯遞送 IP 位址到它的 MAC 位址，ARP 協定在共享的存取網路技術(諸如乙太網路或記號環網路)上使用廣播機制來送出廣播的 ARP 要求訊框。 ARP 回覆會被送回 ARP 要求的發送者，其內包含了所要求的遞送 IP 位址所對應的 MAC 位址。

■ ARP 快取

為了維持被廣播的 ARP 要求訊框在一個最小的量，許多 TCP/IP 協定堆疊加入了 ARP 快取，也就是一個最近被解譯的 IP 位址與其所對應的 MAC 位址之表格。 ARP 快取會在發送 ARP 要求訊框前被檢查。 此外，每張介面擁有其 ARP 快取，如表 2-3 所示。

基於不同廠商的實作，ARP 快取能有以下的能力：

- ARP 快取欄位可以是動態或是靜態的。 靜態 ARP 欄位是永久的且是由 TCP/IP 公用程式(如微軟視窗平台所提供的 ARP 公用程式)所手動加入。 靜態 ARP 快取欄位被用來避免常用地本地 IP 位址的 ARP 請求，例如路由器或伺服器。 靜態 ARP 快取欄位的問題在於當網路介面設備改變時它們必須被手動地更新。
- 動態 ARP 快取欄位有一個對應的時效值用來在當指定的時間到時從快取內移除該欄位。動態 ARP 快取欄位在微軟視窗 TCP/IP 下被移除前所給定的是最長 10 分鐘。

Interface: 192.168.40.123		
Internet Address	**Physical Address**	**Type**
192.168.40.1	00-00-0c-1a-eb-c5	dynamic
192.168.40.124	00-dd-01-07-57-15	dynamic
Interface: 10.57.8.190		
Internet Address	**Physical Address**	**Type**
10.57.9.138	00-20-af-1d-2b-91	dynamic

Table 2 - 3: Windows-based ARP Cache Table

■ ARP 流程

IP 協定傳送資訊給 ARP 協定。 ARP 接收到 IP 封包，遞送 IP 位址與用被用來繞送該封包的介面。 無論採用直接或間接遞送，ARP 協定執行下面的流程，如同圖 2-27 所示：

1. 基於介面與繞送 IP 位址，ARP 協定查閱適當的 ARP 快取，找尋該繞送 IP 位址的欄位。如果該欄位被找到了，ARP 協定跳到步驟 6。
2. 如果該欄位沒被找到，ARP 協定建立一個包含發送 ARP 要求介面 MAC 位址，發送 ARP 要求介面 IP 位址與遞送 IP 位址的 ARP 要求訊框。 接著 ARP 協定便使用適當的介面廣播這個 ARP 要求。
3. 所有的主機接收到這個廣播訊框並處理該 ARP 要求。 如果接收主機的 IP 位址符合所要求的 IP 位址(也就是繞送 IP 位址)，它的 ARP 快取將被更新加入 ARP 要求發送者的位址對應。 而如果接收主機的 IP 位址不符合所要求的 IP 位址，則該 ARP 請求會被丟棄。
4. 接收主機配置一個包含所要求 MAC 位址的 ARP 回覆，並直接傳送給 ARP 要求的發送者。
5. 當 ARP 回覆被 ARP 要求發送者所接收，它會更新它的 ARP 快取加入 ARP 要求與 ARP 回覆的位址對應。 此時，雙方主機在其 ARP 快取內都擁有彼此的位址對應。
6. 該 IP 封包藉由解譯出來的 MAC 位址被送往繞送主機。

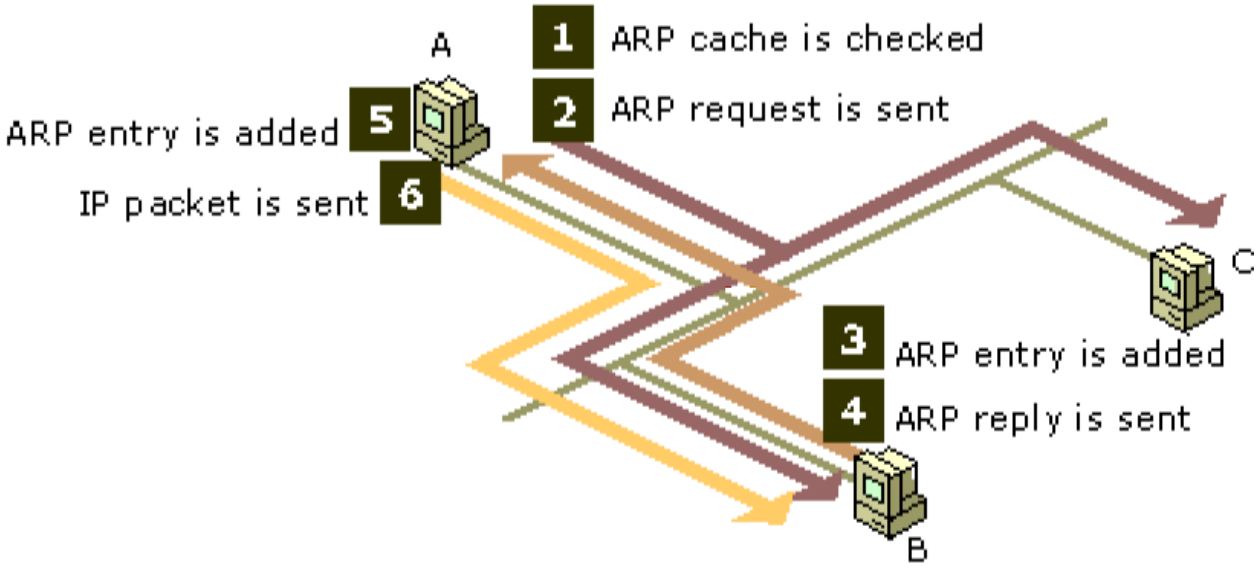

Figure 2 - 27: ARP Process

最後藉由圖 2-28 來回顧一下 TCP/IP 協定堆疊在微軟視窗環境下的網路架構。 這裡必須注意到，無論是 IP 路由或是 ARP 解譯，它們的運作機制都在 IP 協定內完成，也就是在 NDIS 介面之上。 換句話說，當底層媒介為乙太網路時，IP 封包在出 TCP/IP 協定堆疊之後，便形成一個包含完整乙太網路標頭的乙太網路訊框。

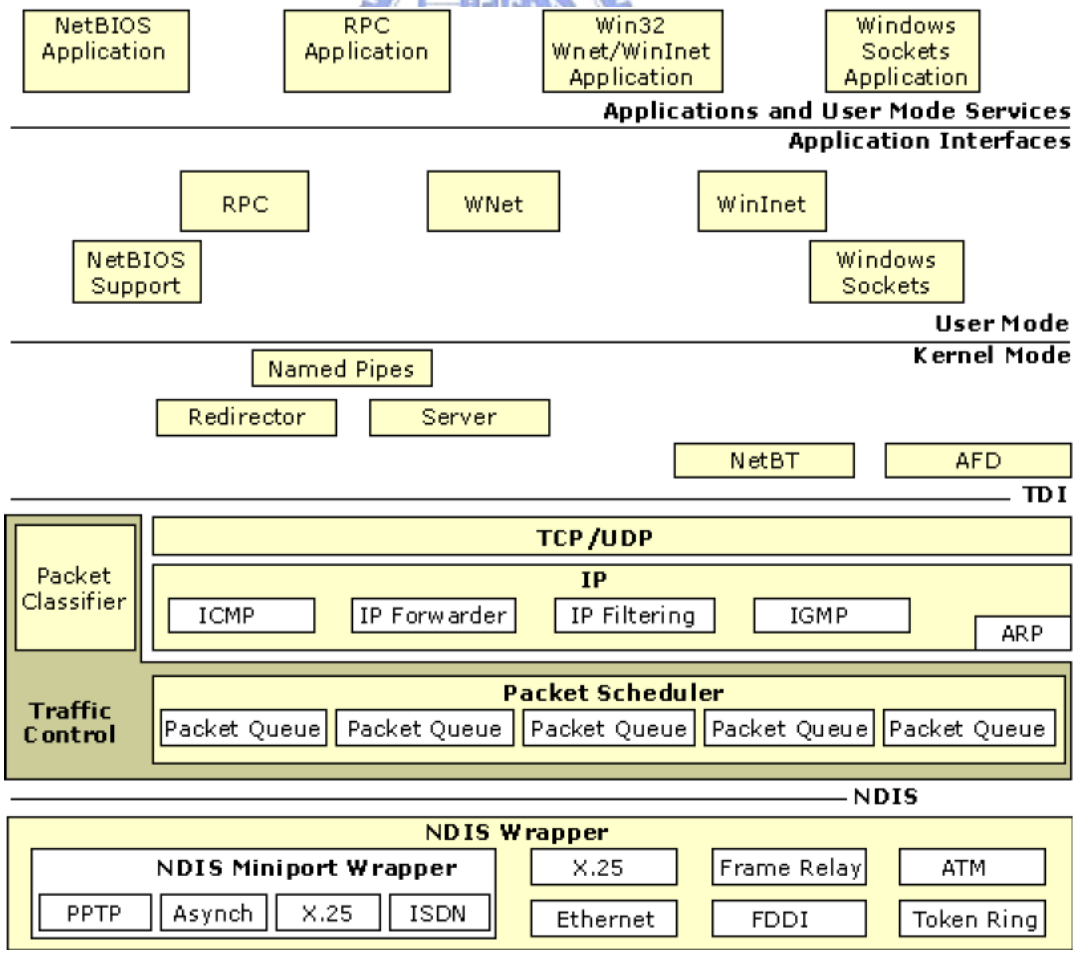

Figure 2 - 28: TCP/IP in the Windows Network Architecture

Chapter 3 Related Work

Section 3.1 Wireless Data Network Integration

近來的趨勢顯示基於 IEEE 802.11 標準的無線區域網路(Wireless Local Area Network, WLAN)與 2.5 代整合封包無線電服務(General Packet Radio Service, GPRS)甚至是第 3 代全球行動電話系統(Universal Mobile Telephone System, UMTS) [20][21]或兩千兆赫的分碼多工存取(Code Division Multiple Access 2000)之廣域無線網路(Wireless Wide Area Network, WWAN)將會共存為使用者提供網際網路的存取。 這兩項技術所提供的特性完美地互補了彼此。 802.11 的標準 [19]使得在視距離基地台(通常稱為存取點，Access Point)的遠近獲取支援從 1Mbps 到 54Mbps 不等速度的便宜無線區域網路。 然而 802.11 存取點能覆蓋的範圍僅僅只有方圓數百公尺，使得它們適合於企業網路以及所謂的熱點(Hot-spot)，例如旅館與機場等。 相反的，使用 2.5 代或第 3 代行動通訊標準所建立的無線網路，需要十分龐大的資金投資。 而所支援有限的尖峰速度從 32Kbps 到最大值幾近 2Mbps，但是卻提供較廣域的覆蓋範圍，使得連線可到處存在。 能讓使用者無縫式地切換於這兩種網路型態間的架構部署將可替服務提供者及使用者帶來多項的好處。 藉由提供整合的 802.11/2.5(3)G 服務、2.5(3)G 營運者與無線網際網路服務提供者(Wireless Internet Service Provider, WISP)能善用他們的資金，吸引更廣大的用戶群並最終促進無所不在的高速無線數據資料之引進。 使用者便能享受到整合服務效能的提升與整體花費的下降。

在本節中將根據歐洲電信標準協會(European Telecommunications Standards Institute, ETSI)所提出的兩種基礎整合架構在 3.1.1 中介紹，並在 3.1.2 探討現有的整合軟體。

3.1.1 Interworking Architectures

在整合異質無線網路上，歐洲電信標準協會制訂了兩種基礎的整合方式，稱為緊密連結(Tight coupling)與鬆散連結(Loose coupling) [24]。 以下將就這兩種架構做基本的介紹：

- 緊密連結整合架構 (Tight Coupling Interworking Architecture)

在緊密連結整合架構之下，可將無線區域網路視為行動電信網路所支援的某一種無線接取網路(Radio Access Network, RAN)。 也就是說無線區域網路將會模擬專屬於行動電信網路才有的功能，以 WLAN 與 GPRS 整合的架構為例，如圖 3-1 所示，圖中的 GIF 便是具備這樣功能的元件。它對行動電信核心的網路隱藏了無線區域網路協定詳細的運作情況，並實作了所有行動電信網路中無線接取網路部分所需要的協定。 而行動端在此架構下需要在標準的無線區域網路卡上實作相對應的行動電信網路協定堆疊，並在需要的時候在實體層做切換。 因此，行動端在無線區域網路下產生的所有封包將會透過行動電信網路協定進入行動電信網路核心再進入網際網路。 換句話說，不論在無線介面卡上所使用的實體層協定為何，這兩種不同的網路將共享相同的認證、通訊、傳輸與計費之基礎建設。 除此之外，採用這樣架構的系統，主要的優點還包括系統可以提供

不斷線的資料傳輸，不需要額外靠 Mobile IP 或其他相關技術才能提供不斷線的資料傳輸。 另外當使用者在行動電信網路與無線區域網路之間切換時，整體程序所需要花費的時間會是最少的。

然而，這個架構存在幾項缺點。 首先是行動電信網路必須在 SGSN(GPRS/UMTS)或 PCF(CDMA2000)上提供新的網路介面來銜接無線區域網路，而開發這個新的網路介面技術並非易事，尤其是要在無線區域網路內模擬或者是轉換行動電信網路中無線接取網路的部分功能，會將原本具有簡單特性的無線區域網路複雜化。 在現實基於資料安全性及保密性的考量以及設定管理的問題下意味著，必須同時營運行動電信網路與無線區域網路的業者才較有機會實現這項架構。 也就是說，獨立營運的無線區域網路除非獲得行動電信網路完整的支援，否則將無法與行動電信網路進行整合。 此外，先前所提到無線區域網路卡必須實作行動電信網路的協定堆疊，這表示在無線區域網路的認證上將採用行動電信網路所特有的通用用戶辨識模組(Universal Subscriber Identity Module, USIM)或可拆卸式用戶辨識模組(Removable User Identity Module) [22] 之認證機制。 這表示無線區域網路卡需有內建的該模組插槽以供放入行動電信網路所提供的晶片卡。

基於上述的原因，由於重新設定行動電信網路核心的複雜度與高成本將會致使選擇緊密連結整合架構的系統商無法與只營運無線區域網路的 WISP 業者競爭。

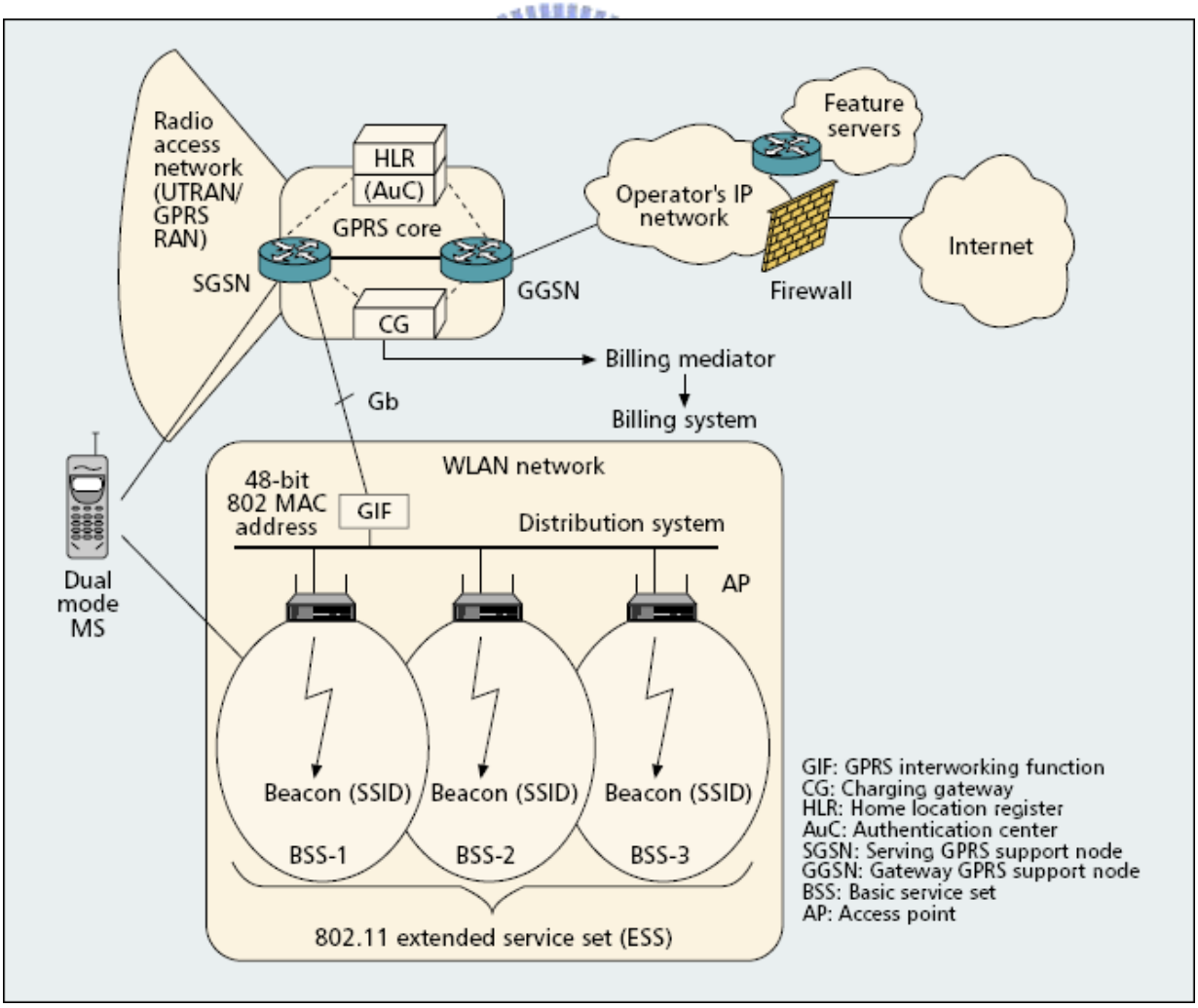

Figure 3 - 1: WLAN/GPRS Integration with Tight Coupling

■ 鬆散連結整合架構 (Loose Coupling Interworking Architecture)

與先前的架構相比較，在鬆散連結整合架構下無線區域網路的資料路徑不會穿過行動電信的核心網路而是直接經過 WISP 本身的 IP 網路或是直接連上網際網路，如圖 3-2 所示。 在此架構下，不同的機制與協定可在各自的網路下用來處理認證，計費與行動管理。 然而，如果要達到不斷線的話，它們必須交互運作。 以 CDMA2000 的狀況來說，在無線區域網路這邊需要 Mobile IP 功能的支援來處理跨網路的行動管理，同時也要有認證授權與計費(Authentication Authorization Accounting, AAA)的服務 [14] 來與 3G 家網路的 AAA 伺服器互動。 而在 UMTS 方面，由於 UMTS 標準尚未支援如 AAA 與 Mobile IP 等的 IETF 協定，所以需要更多的修改來整合 UMTS 網路。 Mobile IP 的服務將必須加入 GGSN 來達成在無線區域網路與 UMTS 網路間的不斷線切換。還有一致的用戶資料庫介面必須被開發使在 UMTS 網路端的 HLR 可以認證與計費，同時也必須讓當用戶漫遊到無線區域網路下時 AAA 伺服器也能執行相同的運作。

採用這種架構的好處是封包不需要經過行動電信網路，路徑簡單快速，也因此所需要行動電信業者的支援程度較輕，比較不受其限制。 對於 WISP 業者來說，擴充性高，系統開發亦較不複雜。 對行動電信業者來說它們能從其他提供無線區域網路佈建者獲取利益而無須擴展其資金設備。 更進一步，當與許多合作伙伴達成漫遊協定所帶來的是更廣大的覆蓋範圍，用戶所受益的是只有一家服務提供者但可享受所有的網路存取。 然而鬆散連結整合架構比起緊密連結整合架構唯一的不足點只在於其行動管理方面。

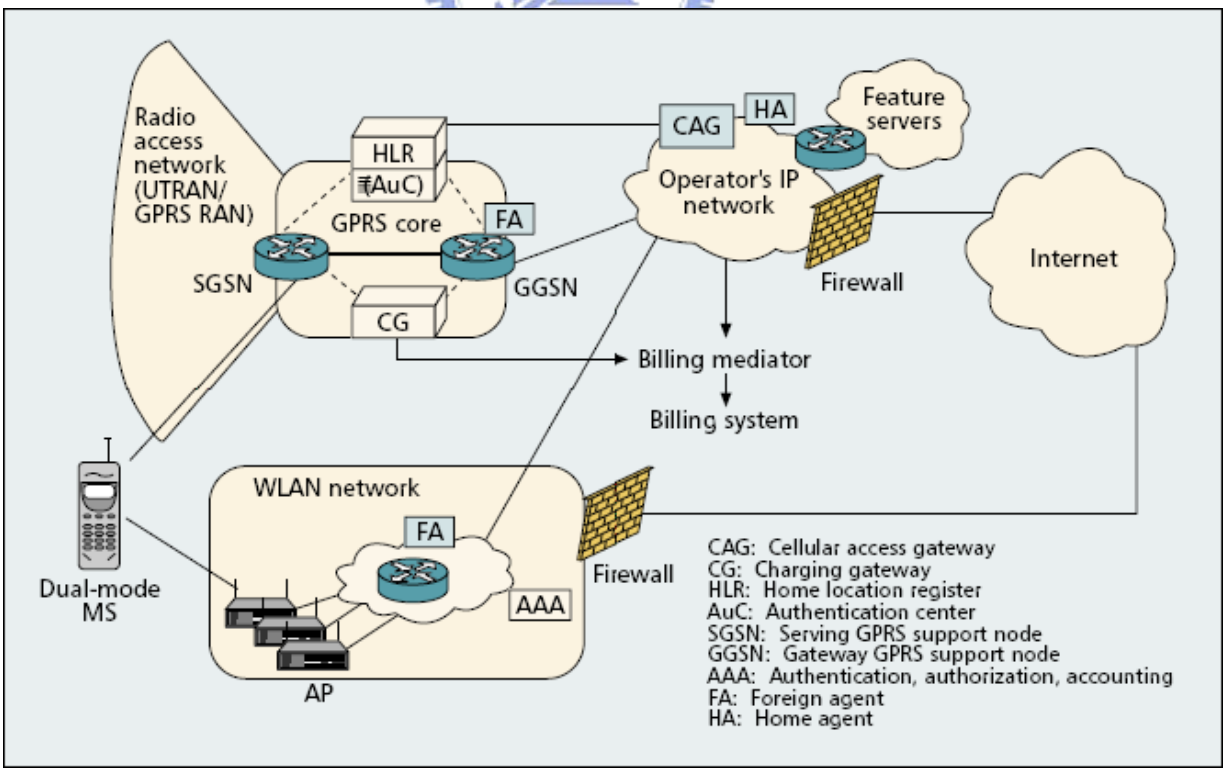

Figure 3 - 2: WLAN/GPRS Integration with Loose Coupling

由以上的分析可知，兩種整合方案各有其優缺點。 但是回頭思索，整合無線區域網路與行動電信網路主要目的之一就是希望利用無線區域網路高速傳輸的優點。 如果像緊密連結整合架構這樣封包都必須經過行動電信的核心網路才能再連上網際網路，這樣便失去整合無線區域網路的意義了。 再加上採行緊密連結整合架構必須大幅受限行動電信業者的態度支持與否，因此現今大部分的系統整合都偏向採用鬆散連結之整合架構。

3.1.2 Currently Implementations

目前針對異質無線網路的整合軟硬體在學界與業界皆有不少的成果發表，本小節內將概略地介紹其中的一些實作。

- NetSwap - University of Portsmouth

NetSwap 為英國普茲茅斯大學名為「無線網路下的不斷線漫遊」(Seamless Roaming Between Wireless Networks)計畫下的產物 [27][28]。 從該系統架構圖(圖 3-3)不難看出這是屬於鬆散連結式的整合架構。 在此架構下主要有兩個元件： NetSwap 驅動程式與 NetSwap 閘道器。在實作方面，NetSwap 的觀念與元件等都是在著名的網路模擬器 ns2 [32] 上實現，並無實際可運作的軟體呈現。 皆下來就針對這主要的兩個元件做介紹：

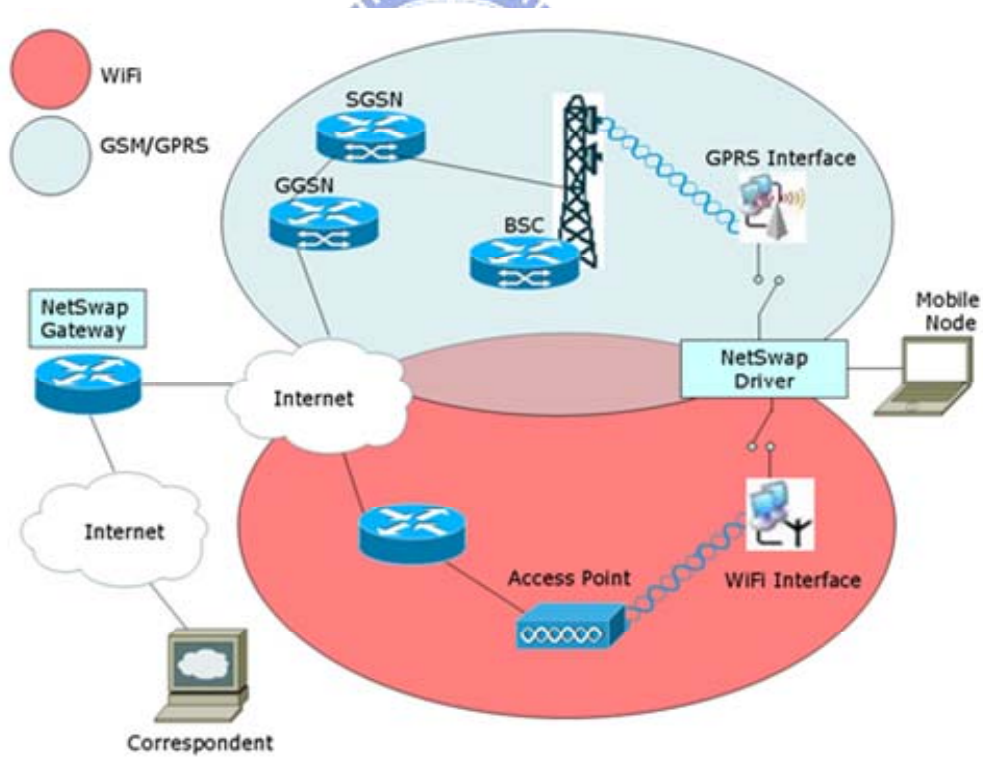

Figure 3 - 3: NetSwap System Architecture

- NetSwap 驅動程式 (NetSwap Driver)

NetSwap 驅動程式是在行動端上呈現出一張虛擬網路介面卡的軟體。 該驅動程式是在所有其它的網路裝置驅動程式之上。 如圖 3-4 所示，其作用是欺騙任何的網際網路應用程式讓它們以為行動端只有一個永遠皆有連線的網路裝置。 NetSwap 驅動程式的運作

就有點像是代理伺服器(Proxy)，視其他不同網路裝置(例如 GPRS 或 WiFi)的存在與否來使用它們。 以第二章所提及的 NDIS 架構來說，NetSwap 驅動程式事實上就是一個中間層驅動程式。

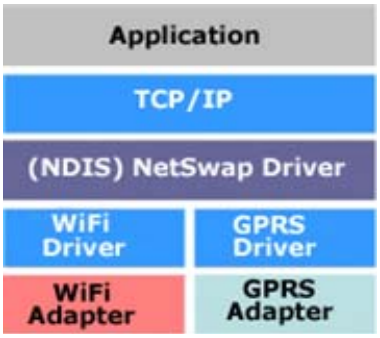

Figure 3 - 4: NDIS NetSwap Driver

NetSwap 驅動程式負責監控所有的連線。 它將來自應用程式的封包根據目前可用與其優先順序每次繞送到一張實體的網路卡 優先順序可以根據花費、路由公制、頻寬或是使用者自訂來決定。 舉例來說，WiFi 的優先權可能比 GPRS 高，因為它的連線速度較快。

要讓任何驅動程式優先使用這張虛擬網路介面卡其中一個作法就是強迫提高系統內其他網路卡的路由公制值，也就是使得虛擬網路介面卡擁有最低的公制。 舉例來說，NetSwap 驅動程式可以給自己的公制為 10，然後更動乙太網路卡的公制值到 11，無線區域網路到 31 以及 GPRS 的值到 51。 如此一來，網際網路應用程式在傳送網路封包時將會透過路由公制值較低的 NetSwap 驅動程式。

接下來，NetSwap 驅動程式必須強迫所有的封包藉由封裝加入第二個 IP 標頭傳送經過一個特殊的路由器(NetSwap Gateway)。 該驅動程式必須要與這特殊的路由器溝通讓它知道目前行動端所使用的介面卡為何。 這樣的通訊在介面卡切換時會以正規的間隔時間被進行。

- NetSwap 閘道器 (NetSwap Gateway)

NetSwap 閘道器是座落於網際網路上的一台路由器，扮演著中繼傳輸所有封包中心點。 它必須擁有一個固定的 IP 位址作為一個大家都知道的點而且該位址不能更動。 而它的工作是遞送所有來自與目的為行動端目前位置的封包。 對於執行在行動端與通訊端上的應用程式來說這是一個永遠存在(Always On)的連線，而且封包通過 NetSwap 閘道器這件事對於兩個端點來說應該要是透明的。

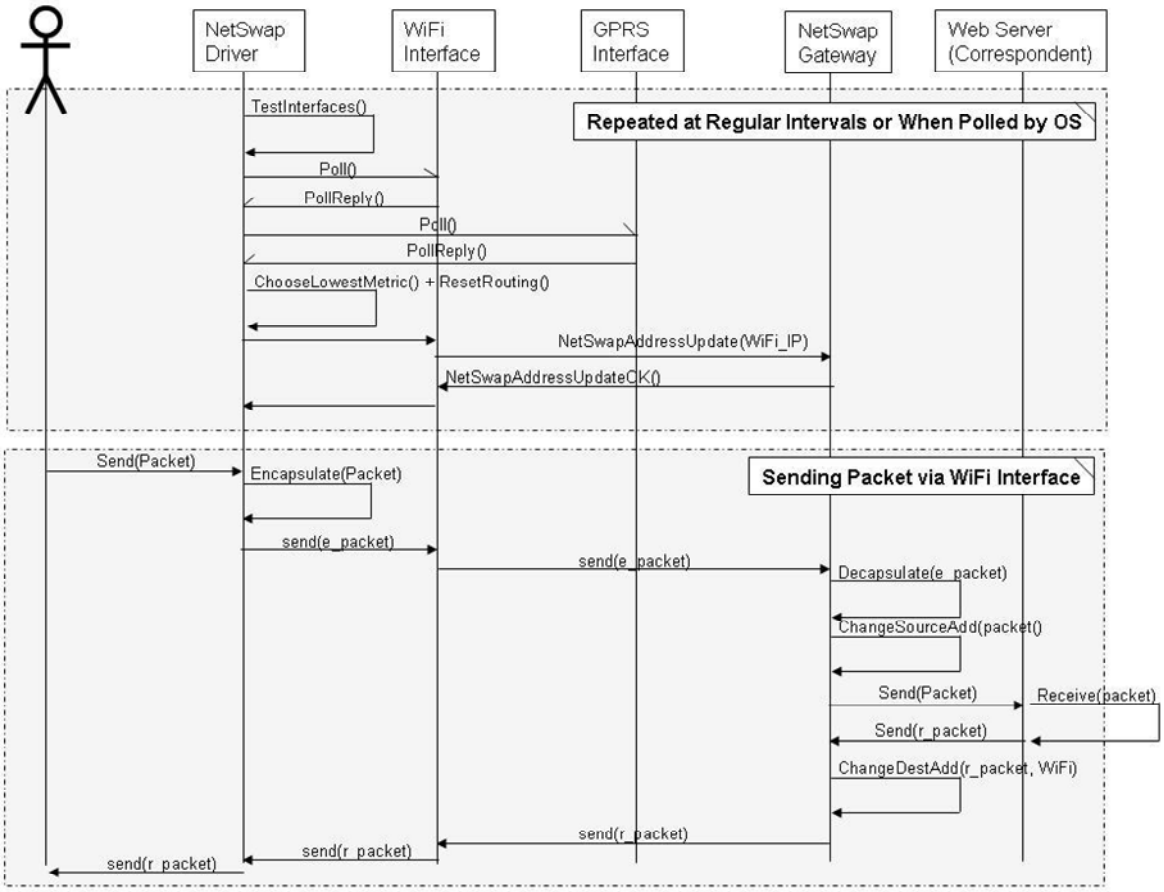

Figure 3 - 5: UML Sequence Diagram for NetSwap

從 3-5 的系統運作流程圖可以更清楚的瞭解 NetSwap 閘道器所扮演的角色與作用。當 NetSwap 驅動程式選定所使用的介面卡，它會利用 NetSwapAddressUpdate()訊息告知 NetSwap 閘道器其決定。 在閘道器確認之後會回送 NetSwapAddressUpdateOK()訊息作為確認，此後當應用程式送出封包時都會經過 NetSwap 驅動程式作適當的封裝後送往該閘道器。 閘道器在收到封包後首先執行 Decapsulate()逆轉封裝獲取原始的封包，接著將原封包運行 ChangeSourceAdd()將封包的來源位址改為 NetSwap 閘道器所擁有的固定 IP 位址，最後將修改過的封包送出。 而在接收到通訊端的回覆封包時，閘道器則執行 ChangeDestAdd()將封包的目的位址改為改為由 NetSwapAddressUpdate()訊息所得知行動端目前所使用介面卡的 IP 位址。 最後便可將封包傳回行動端完成其工作。

以上便是完整的 NetSwap 行動漫遊架構與運作介紹，從以上的介紹中不難發現其實該運作模式非常雷同先前所介紹的 Mobile IP。 只要把 NetSwap 閘道器想成家代理器，NetSwapAddressUpdate()訊息與 NetSwapAddressUpdateOK()想成註冊要求與註冊回覆，那麼兩者簡直是如初一轍。 不過比起 Mobile IP，NetSwap 最大的缺點在於其閘道器同時間僅僅支援一台的行動端，這對於實際的應用上有著極大的限制。 此外，如同 Mobile IP 所遭遇的問題，當

行動端在私有網際網路位址環境下時會無法運作，目前該作者並未針對此問題提出適當的解決方法。

- Vodafone Project - Eindhoven University of Technology

該計畫是荷蘭愛丁荷芬科技大學電子工程所與歐洲著名的電信業者 Vodafone 荷蘭分部所共同合作進行的 [29]。 其研究的重心為不斷線的切換與認證，授權與計費的問題探討。 由完整的架構圖(圖 3-6)也不難看出這同樣是屬於鬆散連結式的整合架構。 此外，從架構圖內也可窺見其整合方式為先前所提及採用配置轉交位址模式的 Mobile IP 協定。 該架構下兩個重要的元件即是家代理器與行動端本身，兩者的軟體皆採用芬蘭赫爾辛基科技大學(Helsinki University of Technology, HUT)所開發的 Dynamics 之 Mobile IP 套件 [33]。 Dynamics 由於其原始碼公開又有完整的說明文件，為目前在 Linux 作業系統環境下最為被廣泛使用的 Mobile IP 套件。 不過可惜的是，它並不支援微軟的視窗環境。 換句話說，此整合的系統被限定運作在 Linux 的環境下。

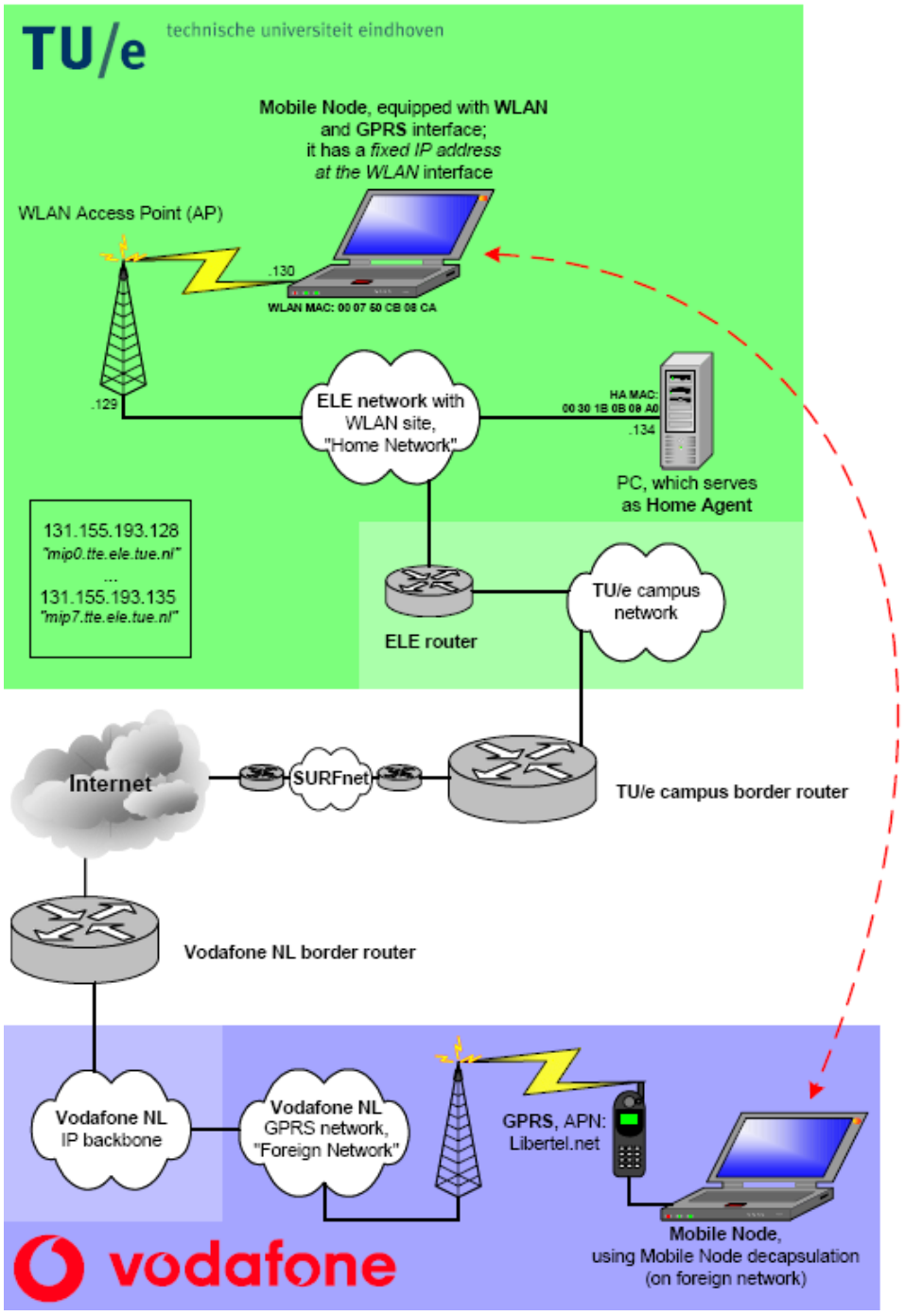

Figure 3 - 6: Dynamics Mobile IP System Architecture

在此架構下的介面不斷線切換是由 Mobile IP 的機制所達成，對於 Mobile IP 的機制已在前面的章節做過完整的描述，接下來將只針對行動端如何切換介面卡做介紹。 在 Linux 環境下，所謂的腳本(Script) 是一種自動完成多種類型工作的方法。 因此在行動端上便有一個腳本被建立用來使用 Dynamics，而該腳本真正的目的則是實作決定可用的介面卡的自動切換程序，如圖 3-7 所示。切換腳本(Handoff Script)內存有每張介面卡的 IP 路由資訊，擁有這些資訊便可控制路由表。 基本上此腳本的運作流程如下：

- 每秒擷取一次目前 IP 路由表的設定並檢視是否有改變。 改變意味著某個 IP 位址，子網路或是預設閘道被更動，譬如說有張介面卡被插入或拔除。
- 一旦它偵測到設定上有所改變，腳本便開始分析並更新到自動選擇或手動指定的偏好介面。
- 最後更新過後的路由表會被再次擷取並回到第一個步驟。

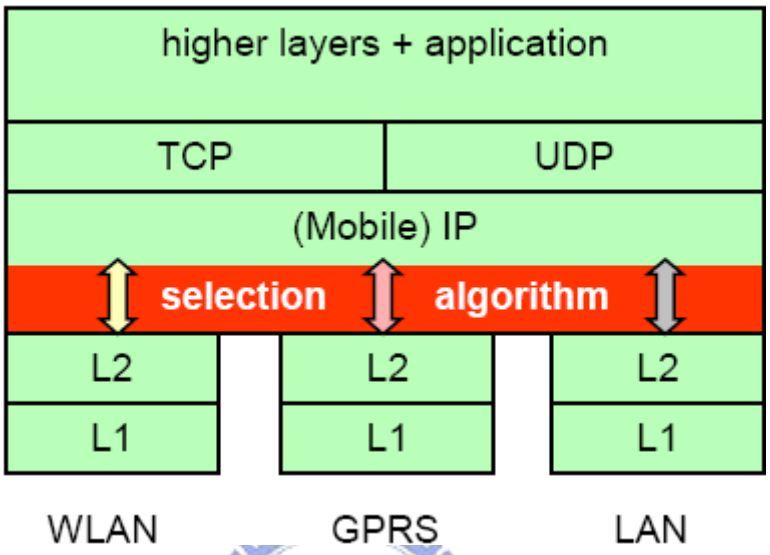

Figure 3 - 7: Handoff Script with Selection Algorithm

上述步驟中所提及的介面更新乃指 Mobile IP 協定中的註冊請求與回覆流程。 藉由使用新選擇的偏好介面 IP 位址作為新的轉交位址送出新的註冊請求，家代理器便會把封包送往新選擇的介面卡，而行動端也會使用該介面卡來收送所有的封包。 如此一來便達到所謂的不斷線切換。 而在偏好介面卡的選擇上，如果採用手動指定模式則優先使用被指定的介面卡，否則將用優先介面卡選擇演算法(如圖 3-8 所示)決定出合適的介面卡。 該演算法的精神是當有兩張同類型的網路卡(例如有線區域網路卡與無線區域網路卡同為乙太網路)存在時優先使用新加入的網路卡，而乙太網路介面卡的優先權高於點對點配接卡(如 GPRS, PHS, CDMA 2000 等)。 由於該演算法過於簡單，未加入網路特有性質的考量，諸如訊號強度、頻寬與花費等，是本實作最需改進之處。

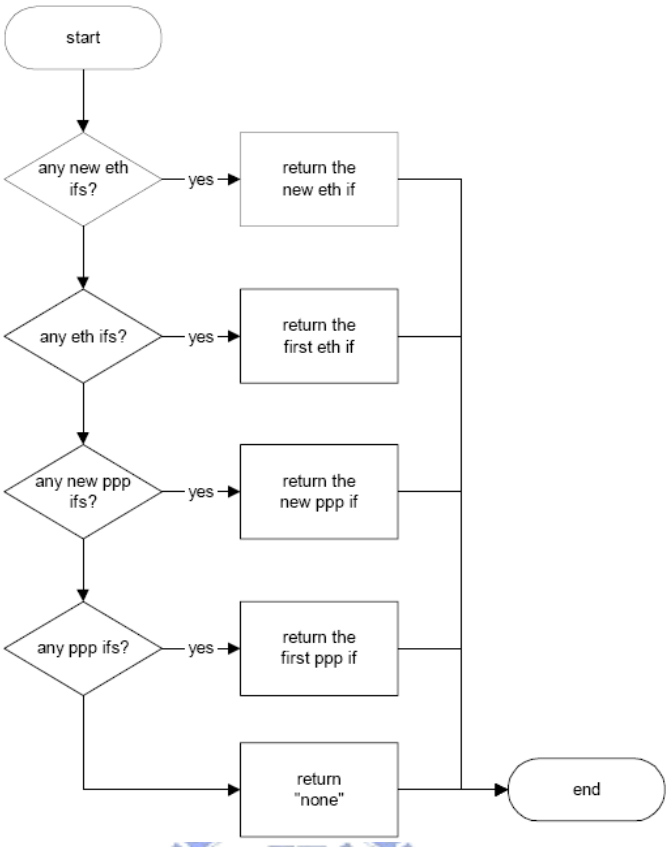

Figure 3 - 8: Priority Interface Determination Algorithm

■ IOTA Project – Bell Labs, Lucent Technologies

著名的貝爾實驗室在其名為整合兩種存取科技(Integration Of Two Access Technologies, IOTA)計畫下也開發了一套以鬆散連結式整合架構所整合的系統 [25][26]。 該系統同樣有兩個主要的元件，IOTA 閘道器與 IOTA 客戶端。

- IOTA 閘道器 (IOTA Gateway)

IOTA 閘道器整合了數個子系統，如圖 3-9 所示： 認證授權與計費之 Radius 客戶端與伺服器，Mobile IP 之外地代理器與家代理器、DHCP 伺服器、NAT 模組、QoS 模組與整合的網頁模組等。 視不同的硬體設定，閘道器可以有內建的無線區域網路存取點或是外接式的。 而閘道器的軟體皆跑在運作 Linux 作業系統的主機上。

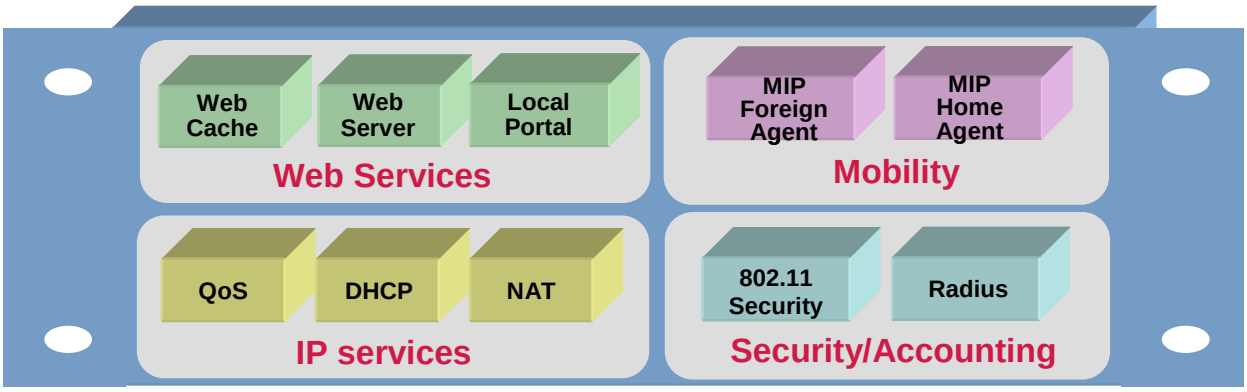

Figure 3 - 9: IOTA Gateway Software Architecture

- IOTA 客戶端 (IOTA Client)

IOTA 的客戶端是實作在微軟視窗 XP 的平台之上，如圖 3-10 所示。 主要可分為三個部分： 圖形化使用者介面(Graphical User Interface, GUI)，在用戶模式下的行動端工作與在作業系統核心網路協定堆疊底層的裝置驅動程式。 用戶模式下的工作包括完成完整的 Mobile IP 堆疊與運行大部分的行動管理。 而驅動程式則提供了一個抽象的虛擬網路卡給系統的協定堆疊。 也就是對於應用程式來說，虛擬網路卡隱藏了行動的相關細節。 圖形化使用者介面則允許使用者設定，監控與控制用戶端的狀態。 同時藉由在 Mobile IP 之上運作 IPSec，該系統也可支援虛擬私人網路(Virtual Private Network, VPN)的運作。 目前此客戶端只可搭配 Lucent 公司所開發的 IPSec 客戶端來使用。

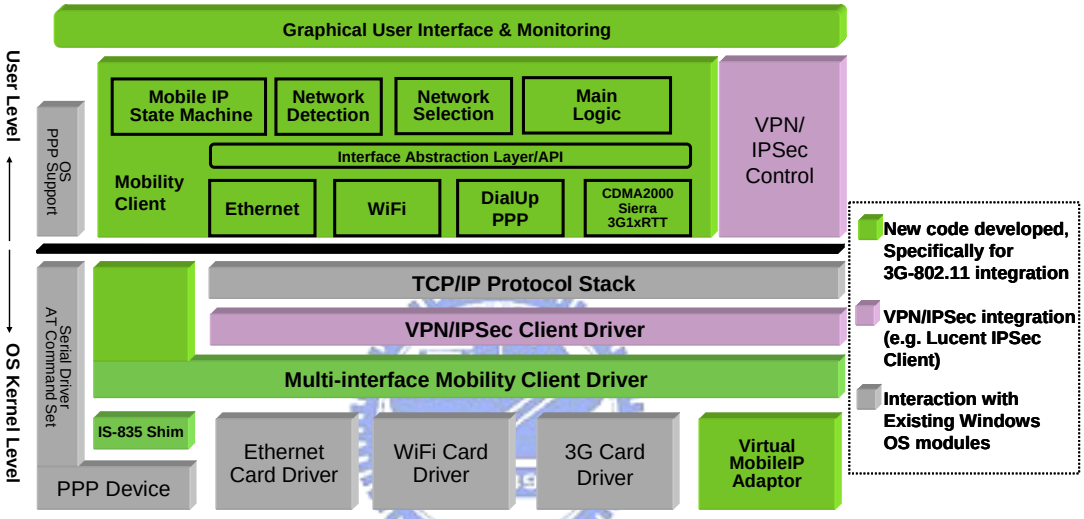

Figure 3 - 10: IOTA Client Software Architecture

除此之外，IOTA 在客戶端的實作上還設計了一個介面卡的選擇演算法。 該演算法使用了目前網路卡的訊號強度與優先權來選擇所使用的介面卡。 同時它也避免了當兩張網路卡訊號其度相接近時所發生不必要的震盪。 演算法中有四個主要的變數： 正規劃的訊號強度(s)、優先權(p)、最低門檻值(L)與最高門檻值(H)。 如圖 3-11 所示，我們以 si、pi、Li 與 Hi 來表示介面卡 i 的各項變數值。 而 Li 與 Hi 的值在 0 與 100 之間，pi 可能的值為 1,2 與 3。 客戶端週期性以此演算法計算出每張介面卡 i 的權值 wi，並切換到有最高權值的介面卡。

If i is the current interface,

$$w_i = \begin{cases} 1000 * p_i + 2s_i & \text{if } s_i \geq L_i \\ 2s_i & \text{if } s_i < L_i \end{cases}$$

If i is not the current interface,

$$w_i = \begin{cases} 1000 * p_i + s_i & \text{if } s_i \geq H_i \\ s_i & \text{if } s_i < H_i \end{cases}$$

Figure 3 - 11: IOTA Client Interface Selection Algorithm

Section 3.2 Mobile IP NAT Traversal

在 2.1.4 節中曾提及 Mobile IP 協定運作於私有網際網路位址環境下會遭遇到的問題，在本節中將回顧此問題並介紹現有的解決方案。 首先，如圖 3-12 所示，如果行動端的家代理器位在公有網路上，而當行動端漫遊到私有網路內時便會發生問題。 圖中的 NAT/NAPT 路由器擁有兩個 IP 位址，一個是對內私有網域的 10.113.215.254，一個是對外公開網路的 140.113.24.68。 當行動端漫遊到此私有網域下獲得一私有 IP 位址 10.113.215.188 後，透過該路由器向家代理器註冊。此時由於 Mobile IP 的註冊請求訊息是透過 UDP 封包來搭載，因此在通過 NAT/NAPT 時會在其上的位址轉譯表新增一個欄位，紀錄由 10.113.215.188 主機的 UDP 埠號 434 送出的封包將被轉譯成來源 IP 位址為 140.113.24.68 來源埠號為 54792。 而當家代理器回送註冊回覆訊息到來源位址與埠號，也就是該 NAT/NAPT 路由器，由於註冊回覆訊息也是由 UDP 封包來搭載，因此經查詢位址轉譯表後發現送到 140.113.24.68 的 UDP 埠號 54792 將被轉譯成送往 10.113.215.188 的埠號 434。 因此註冊請求與回覆訊息可順利的通過位址轉譯器完成 Mobile IP 的註冊，然而，對於接下來的 IP 承載 IP 封裝通道建立就並非如此了。 由於 IP 承載 IP 封裝方式不會有埠號之資訊，當該封包送達 NAT/NAPT 路由器時會因缺乏轉譯的根據而封包遭到丟棄。 所以縱使行動端可在私有網際網路位址環境下向家代理器註冊，但卻無法在該環境下收到封包資料。

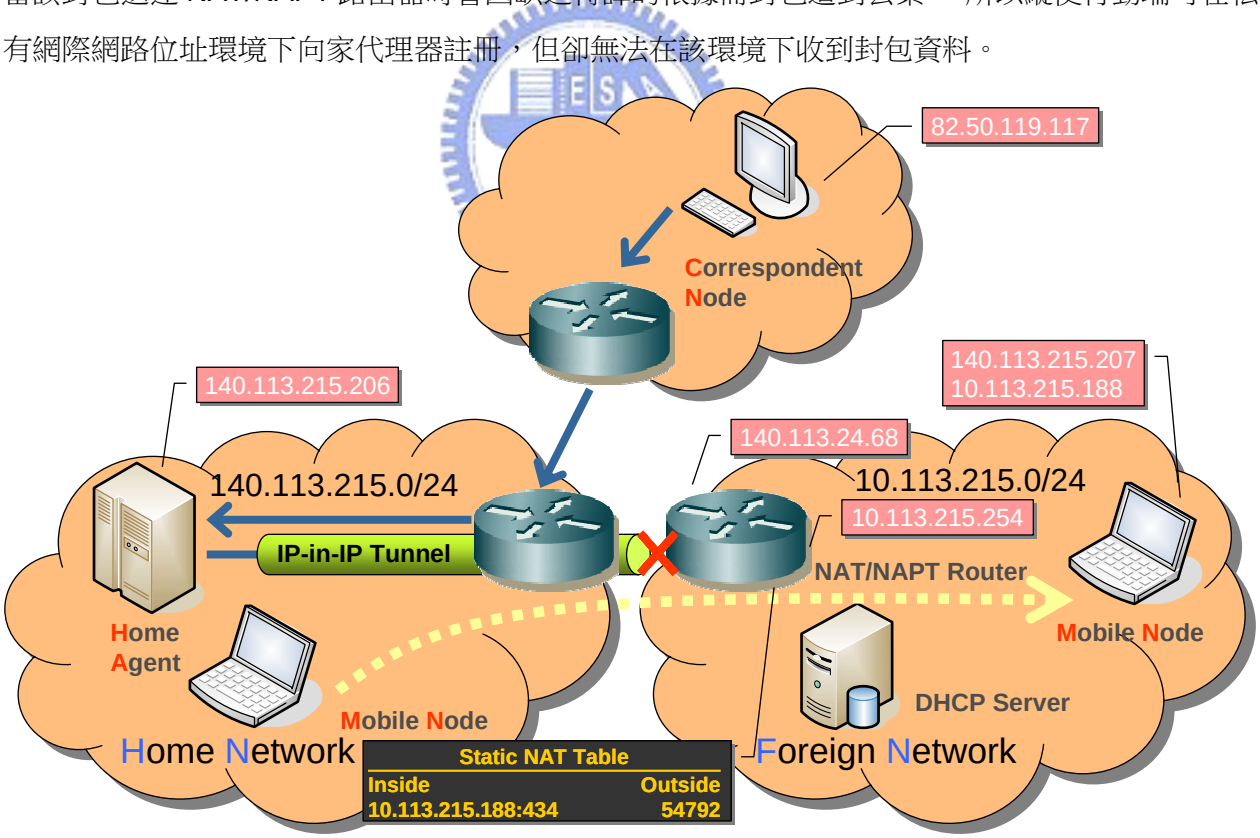

Figure 3 - 12: Mobile IP NAT Traversal Problem

以上便說明了為何原有的 Mobile IP 協定無法運作在私有網際網路位址的環境下。 接著將介紹現有的解決方案，分別是 3.2.1 的美國專利以及 3.2.2 的 RFC 3519。

3.2.1 The Patent Solution

美國專利編號 US2003/0123421 A1 文件中提出了一個在 Mobile IP 協定裡穿越網路位址轉譯器的方式。 該方法相當的簡單，也就是運用 2.2 節中所提到的動態網路位址轉譯器，如圖 3-13 所示。 透過一對一的位址轉譯，IP 承載 IP 封裝通道便可以順利的通過網路位址轉譯器到達位在私有網路下的行動端。 然而此方法最大的缺陷就是網路位址轉譯器必須擁有多個公開網際網路位址才可以服務在私有網路下一個以上的行動端。 不過這樣一來便違背最初網路位址轉譯器被設計的用意。

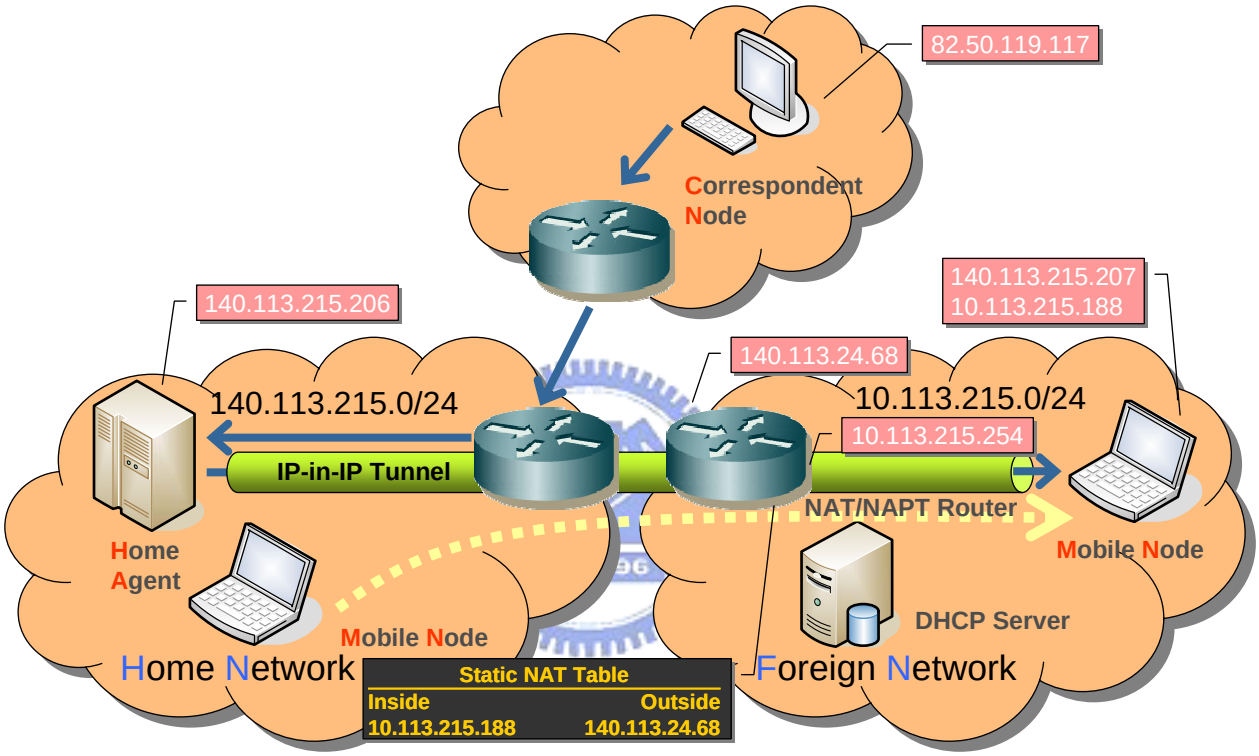

Figure 3 - 13: US Patent Mobile IP NAT Traversal Scheme

3.2.2 The IETF Solution

IETF 組織在 RFC 3519 文件內制訂了標準的 Mobile IP 協定穿越網路位址轉譯器的方法[17]。 由上述的問題描述可以發現是 IP 承載 IP 的封裝方式造成 Mobile IP 協定無法於私有網路環境下運行，因此在該文件中便引進了一種新的封包封裝方式，稱為 UDP 承載 IP 封裝。 如圖 3-14 所示，該封裝方式的用意是補足原有 IP 承載 IP 封裝方式通過網路位址轉譯器時所缺少的埠號資訊。 因此，在正常的情況下，行動端透過網路位址轉譯器向家代理器註冊後，會在其上建立一組 IP 位址與埠號轉譯對應。 UDP 承載 IP 封裝便是利用註冊流程所使用的埠號來建立通道，舉例來說，承接問題描述中的例子，當通訊端 82.50.119.117 所傳送給行動端的封包被家代理器接收後，會在其外加上目的位址為行動端註冊請求的來源位置，也就是網路位址轉譯器的公開位址 140.113.24.68 之 IP 標頭，還有目的埠號同為註冊請求的來源埠號 54792，最後加上一個型態欄

位與次標頭欄位同為 4 的 Mobile IP 標頭(型態 4 表示 UDP 所後面所承載的是 Mobile IP 之通道資料，次標頭 4 表示通道資料內為一由 IP 標頭所承載的封包)。 如此一來，以 UDP 承載 IP 封裝的封包到達網路位址轉譯器時，便可在其內的位址轉譯表查詢到註冊流程時所建立的對應關係，該封包便可順利的透過位址與埠號轉譯送達行動端，見圖 3-15 所示。

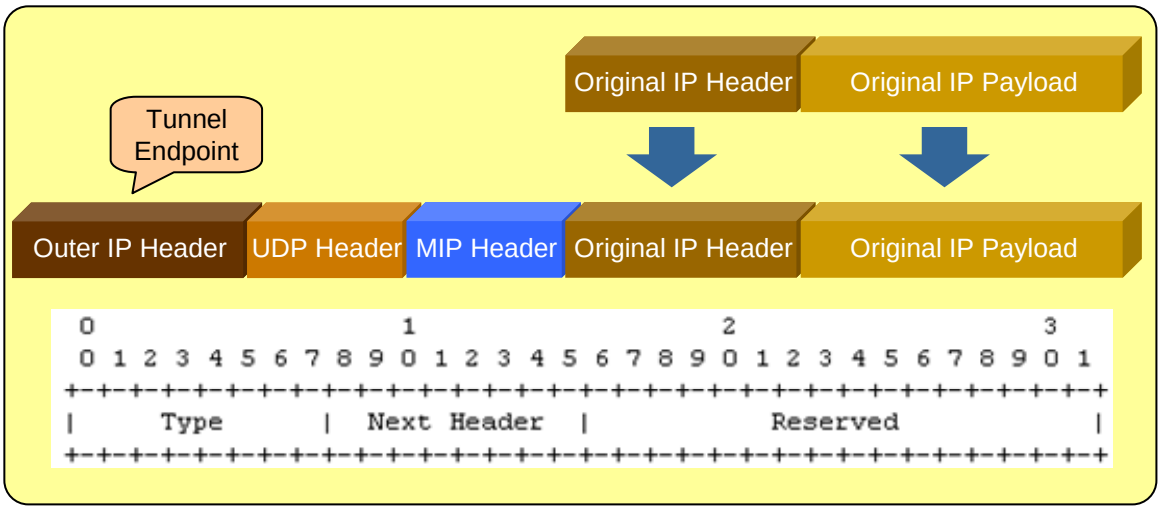

Figure 3 - 14: IP-in-UDP Encapsulation and Header Format

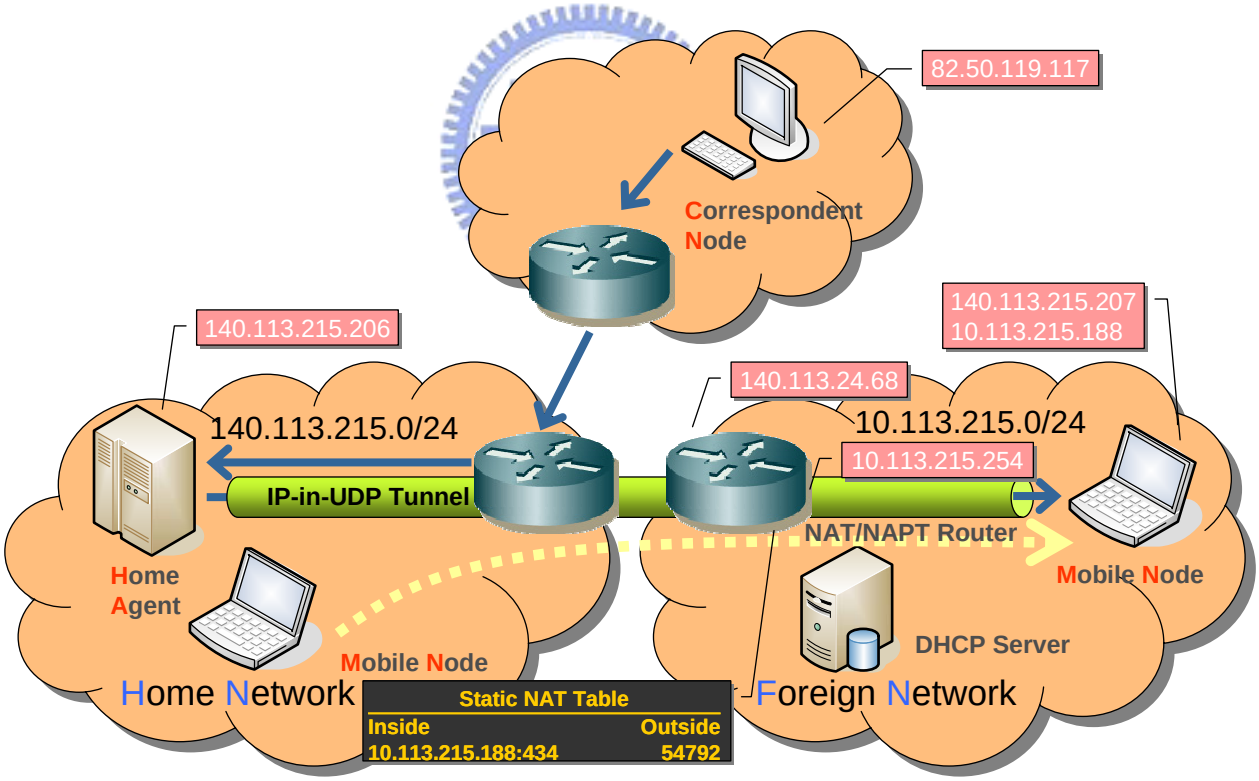

Figure 3 - 15: IETF Mobile IP NAT Traversal Scheme

此外，為了讓行動端與家代理器協商 UDP 承載 IP 封裝的使用，文件中在原有的 Mobile IP 協定裡新增了分別用在註冊請求與註冊回覆所使用的延伸，如圖 3-16、3-17 所示。 它們的使用方式就如同 2.1.2 中所描述，必須放在註冊訊息之後驗證延伸之前。

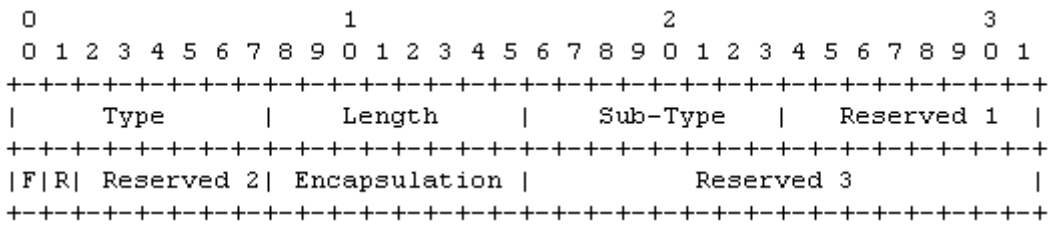

Figure 3 - 16: UDP Tunnel Request Extension Format

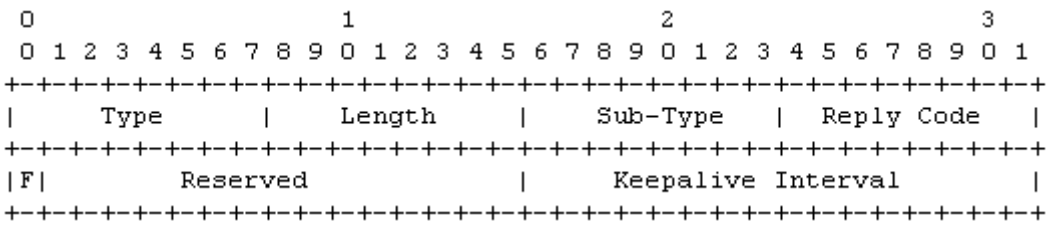

Figure 3 - 17: UDP Tunnel Reply Extension Format

由 IETF 所提出運用 UDP 承載 IP 封裝成功地克服了 Mobile IP 行動端漫遊到私有網際網路位置的外地網路下所遭遇到的問題。 目前該解決方案被廣泛的使用在 3.1.2 中所提及各種以 Mobile IP 協定作為不斷線切換的整合系統實作上。 然而相較於原有的 IP 承載 IP 封裝方式，UDP 承載 IP 封裝在額外新增標頭部分的負擔明顯呈現較大的成長。 這樣的現象可能造成的是封裝後超過最大傳輸單位(Maximum Transmit Unit, MTU)，封包便可能在傳輸的過程中遭遇到被多次分割(Fragment)與重組，使得傳輸的效能下降。 此外，越多的額外標頭表示真正可用來傳送資料的空間就被壓縮了，對於頻寬較小的媒介來說，所影響的是有效的頻寬運用率下降。 針對這些缺失，在稍後的章節將會提出一改善的方法。

Chapter 4 Radio Mobile IP (RIOMIP)

Section 4.1 Objectives

RIOMIP 是用來達成第三章所提及以鬆散連結的方式整合各種異質網路媒介的客戶端軟體。主要的目的是以 Mobile IP 協定達成在微軟視窗平台下切換使用各種異質媒介網路卡時的不斷線漫遊服務。 取名 RIOMIP 的用意是指如同無線電(RadIO)本身的特性擁有多個頻道可供切換使用，而我們所開發的軟體便是在底層擁有多張可供切換的介面卡上運行 Mobile IP 協定所提供的不斷線漫遊服務。 其餘 RIOMIP 的主要特性與預期達到的功能如下所列：

- 遵守 IETF 相關文件之 Mobile IP 行動端實作

由於我們只實作行動端本身，所以勢必要搭配其它行動代理器來使用。 因此為了彼此的互通性，在 Mobile IP 協定的實作部分一律遵照 RFC 3344 文件的規定並達成對於行動端部分的最小需求。 此外，考量到現今環境下並非每個網路環境皆可放置外地代理器，因此目前版本的實作只支援配置轉交位址模式。

- 支援各種型態之乙太網路卡

我們所開發的軟體套件並不限定使用者使用何種廠牌與型號的網路卡，只要對於系統來說是一張乙太網路型態的介面卡即可。 乙太網路卡包括最常被使用的有線區域網路卡，無線區域網路卡以及使用撥號方式連線的 GPRS、PHS 數據卡等。 當然各種介面 PCI、USB、PCMCIA 甚至是 COM PORT，不管內建或外接，皆不影響系統的運作。 不過前提是這些裝置都必須將其驅動程式正確的安裝後方可使用。 此外，在目前版本的設計上，為了切換的需求，因此必須有兩張以上的乙太網路卡程式才可正常運作。

- 允許行動端漫遊於具備網路位址轉譯器的私有網路內

在支援私有網路方面為了與現有行動代理器配合，同樣遵守 RFC 3519 所訂立的協定。 也就是在 3.2.2 所提及的方法。

- 自動偵測目前系統可供使用之網路介面

為了降低使用者與程式的互動，在設計上我們將網路卡分為主卡(Master)與副卡(Slave)。 程式會自動檢視其是否可用自動地做切換，舉例來說，當主卡因網路斷線無法使用時，程式便會自動程式目前有連線的副卡，當主卡網路連線恢復時，會再度切換回主卡。 當然我們也提供介面供使用者做手動的選擇。

- 提供底層網路卡相關資訊及時回報機制輔助介面卡切換

由先前 2.3.1 的介紹可知道在微軟視窗環境下的網路分層架構中，只有在 NDIS 元件內的驅動程式可以獲取底層網路卡的相關資訊。 對於一般用戶模式下的應用程式來說便無法存取這項資源，然而其中某些資訊，例如無線區域網路卡的訊號強度以及是否連線等資訊，對於在網路的切

換上有莫大的幫助 [18]。 因此我們的實作將達成讓應用程式也可共享這些資訊，此外為了將系統的負擔降到最低，在非必要使用輪詢(Polling)的方式時將盡量採用較即時的回呼函示來實作這項功能。

- 元件組合式軟體架構

在實作系統這項元件時將採用元件組合方式，也就是獨立開發每個元件，再加以整合。 所以每個元件皆可拆開獨立使用，如日後要更換或是增強某部分元件，並不會牽動系統內的其他部分，使得系統維護與更新更為容易。

- 無須修改原有系統核心與網路元件

微軟視窗作業系統下不管是核心或是網路元件的部分多半是未公開的，倘若使用未文件化的 API 來修改原有的元件可能會造成系統無法預期的錯誤使得開發更加困難。 所幸視窗環境的設計架構皆是階層化的，也就是遵照公開文件的規範便可在原有的架構下插入新的元件。 因此在我們的設計上皆遵照此原則來確保系統原有的穩定性。

- 不改變現有各種網路架構與終端裝置

在本系統的整合架構中，無須修改現有網際網路環境的元件包括路由器，網路位址轉譯器，DHCP 伺服器等以及電信網路內的各項元件。 唯一可能需要更動的是為了安全性考量的防火牆設定，其必須開放讓 Mobile IP 協定的封包穿越方能使系統運作順利。

Section 4.2 System Architecture

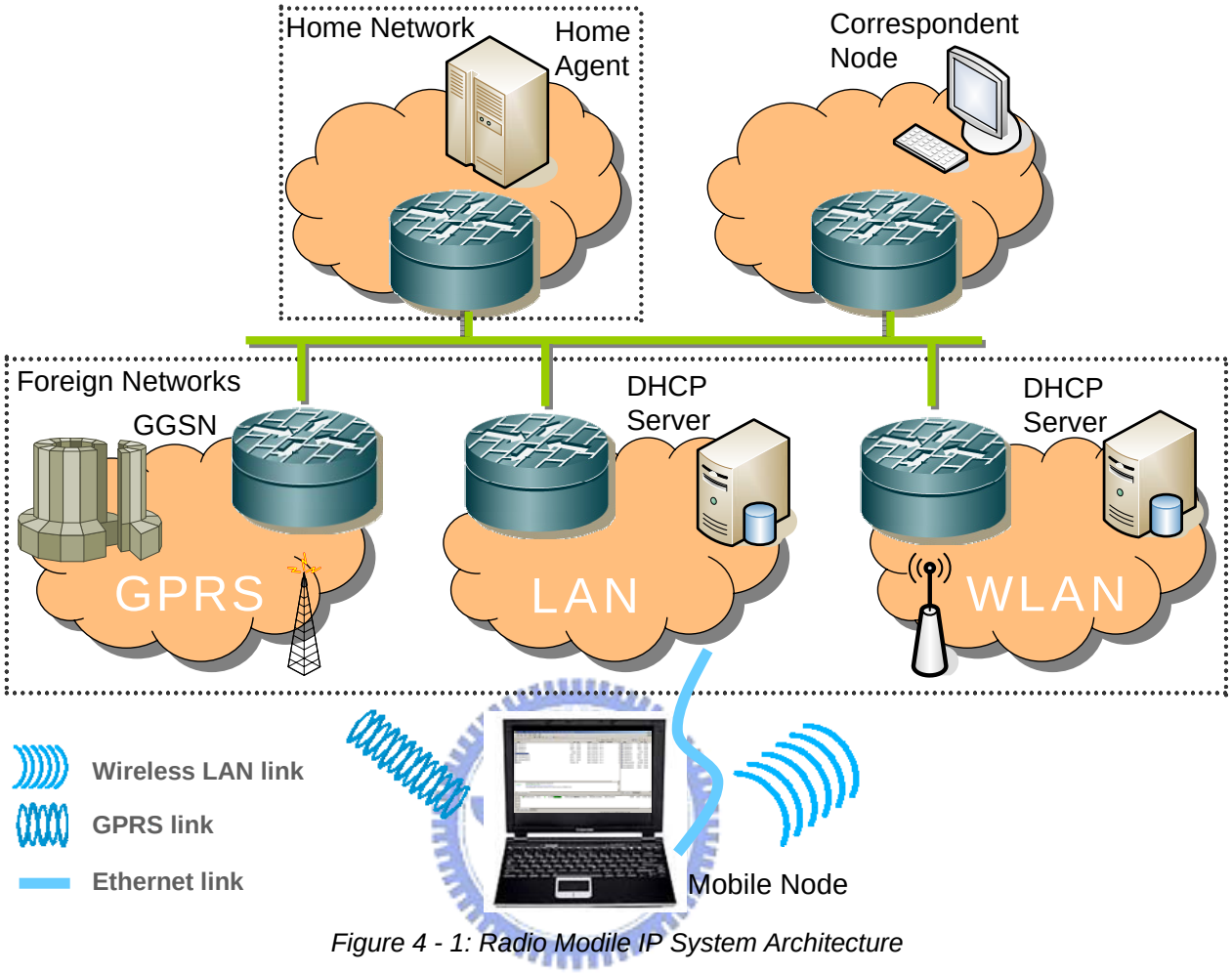

Figure 4 - 1: Radio Modile IP System Architecture

圖 4-1 所展示的便是系統架構的示意圖。 最下方的行動端為一台運行微軟視窗作業系統(2000 或是 XP)的筆記型電腦，並有內建的乙太網路卡可供連上區域網路(圖中中間所示)，此外還有內建/外接之 802.11a/b/g 網路卡透過存取點連上無線區域網路(圖中右方所示)或是插有 SIM 卡的外接式 GPRS 數據卡透過基地台經過行動電信網路接上網際網路(圖中左方所示)。 RIOMIP 程式便是跑在此台電腦上。 上述三個行動端可能漫遊到的網路也就是先前所提及的外地網域，為了不更動或新增元件在該網路下(由於並非所有的網路都由我們所掌控，例如行動電信網路便是例外)，我們採用 Mobile IP 的配置轉交位址模式。 也就是說在外地網域內皆存在 DHCP 服務，在區域網路與無線區域網路下是由 DHCP 伺服器負責，而在行動電信網路則是 GGSN。 當然行動端所得到的位址有可能是公開或是私有的網際網路協定位址。 而在圖中左上方的家網域內有家代理器負責處理行動端漫遊時的行動管理與連線維持。 其上所使用的軟體為芬蘭赫爾辛基科技大學所開發的 Dynamics 之 Mobile IP 套件並搭有由本實驗室所開發支援 RFC 3519 的功能之更新。 行動端可在漫遊時與上述網路之外的通訊端(圖中右上方)建立連線，例如以 FTP 協定傳輸資料等。 此外我們

所預期的是該連線不會因網路的切換而中斷，只有傳輸速率可能因不同的網路而有所改變。 最後我們假設(事實也很有可能如此)上述所有網路的路由器皆搭載在 2.1.4 所提及的進出過濾器。

Section 4.3 Roaming Scenarios

在本節中將介紹以 RIOMIP 整合異質網路的應用實例，分別是 4.3.1 的有線區域網路與無線區域網路間之漫遊與 4.3.2 的無線區域網路與整合封包無線電服務間之漫遊。 在這兩個實例內，我們分別會以圖例解說漫遊之情境後再細看漫遊時的訊息流程。

4.3.1 LAN-WLAN Roaming Scenario and Message Flow

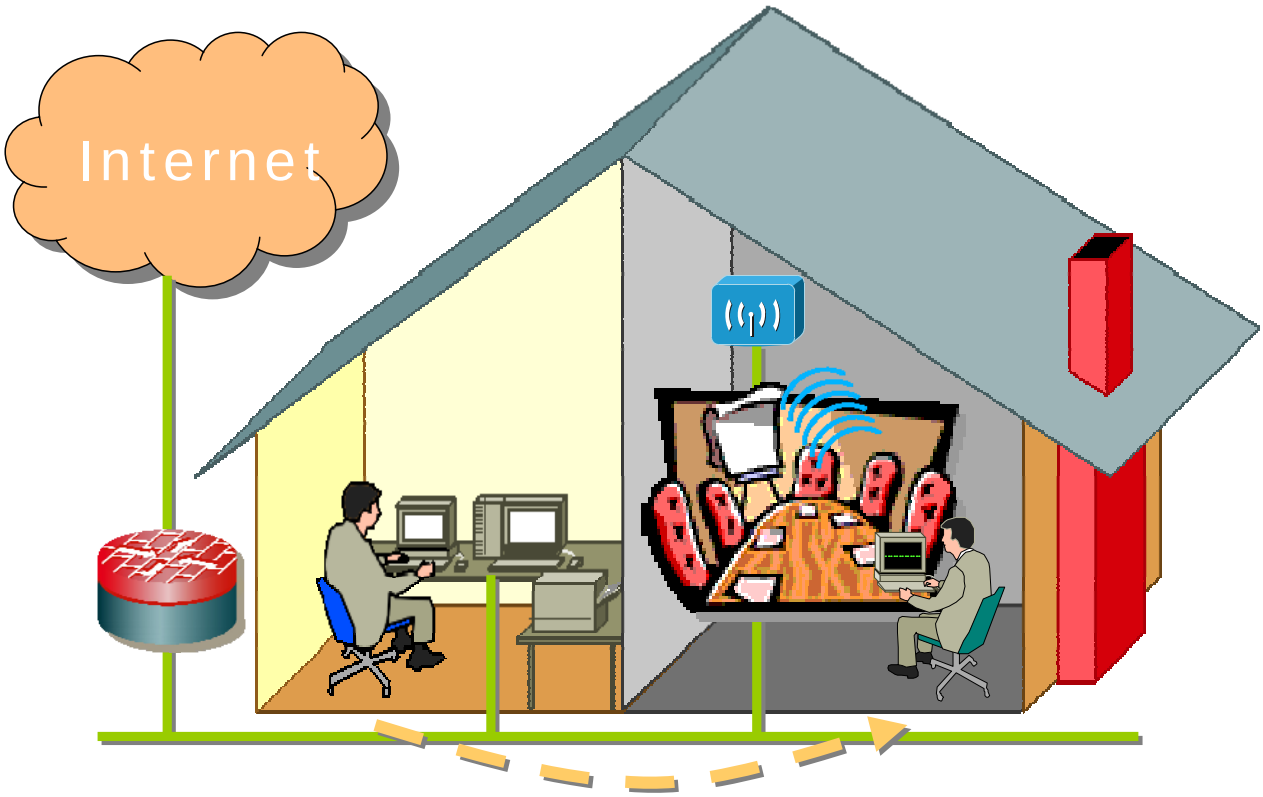

Figure 4 - 2: LAN-WLAN Roaming Scenario

圖 4-2 中的主角名為威爾坑，他是一位亞瑟科技公司的電腦軟體工程師。 平時除了每天下午的例行性會議外他都會在自己的辦公室內以網路線接上他工作用的筆記型電腦連上公司的網路處理相關的事務。 今天由於事務繁忙，所以趕在開會的前一刻他才在自己的辦公室內開始傳送待會會議進行中所需的文件到公司的網路印表機。 然而由於時間緊迫，他無法等候至傳送結束就必須帶著他的筆記型電腦趕赴開會的會議室。 此時，他最不願意見到的事情就是傳送到一半的檔案中斷，他所樂見的是在他移動前往會議室的途中，也可以透過英代爾所標榜的迅馳(Centrino)行動運算技術筆記型電腦連上公司內部的無線區域網路持續傳送未完成的檔案。 所以當他到達會議室以後就可安心的參與會議，列印好的文件就交由美麗的秘書小姐處理。

上述的情境，便是運用 RIOMIP 來處理網路間不斷線切換的一個實例。 接下來我們便檢視一下在該情境下所產生的訊息流程：

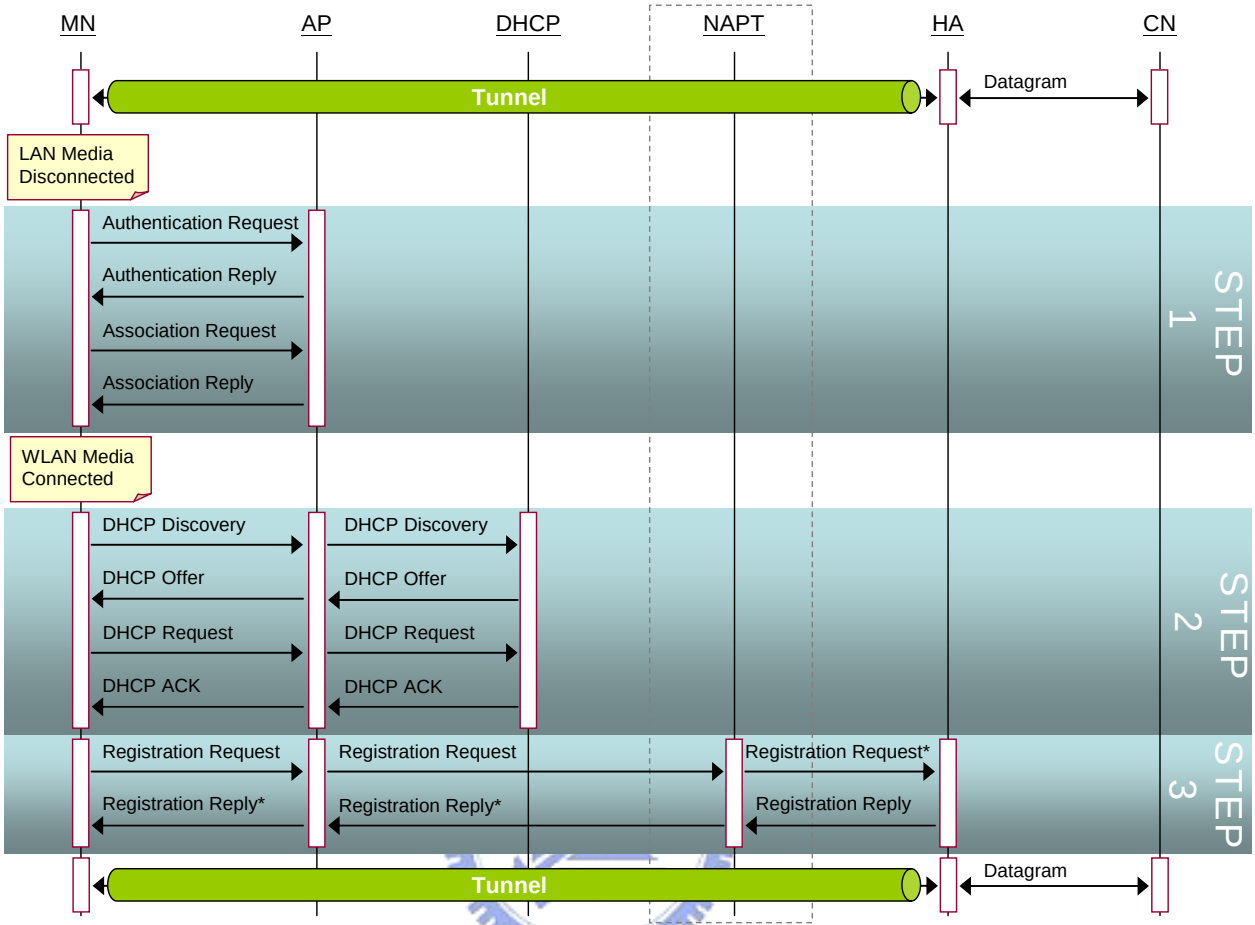

Figure 4 - 3: LAN Handoff to WLAN Message Flow

當威爾坑工程師在起身離開辦公室時，他拔除了原先插在筆記型電腦上的網路線。 此時，底層網路卡驅動程式會觸發一已拔除網路電纜之訊息告知上層其它的驅動程式，而該訊息也同時會被 RIOMIP 程式所擷取。 如圖 4-3 所示，當區域網路電纜已拔除的訊息產生後，原有的傳輸連線會暫時停止，此時 RIOMIP 程式會試圖使用無線區域網路卡來維持原有的連線。 如圖中的步驟一，首先無線區域網路卡會遵照 802.11 協定的規範與在其連線範圍內的存取點建立連結，所使用的訊息包括認證請求與回覆以及連結請求與回覆，當然如果在此之前已完成此步驟便可省略。 當步驟一完成後無線區域網路卡的驅動程式會觸發網路已連線的訊息，接著該訊息被 TCP/IP 協定堆疊接收到後會啟動步驟二 DHCP 的運作以取得 IP 位址。 而 RIOMIP 程式內則有一專門負責監控系統 DHCP 運作的元件，一旦發現網路卡取得 IP 位址或是原有的位址發生變更，便會要求負責 Mobile IP 的元件送出註冊請求的訊息，這部分也就是圖中的步驟三。 當 RIOMIP 收到成功的註冊回覆訊息後，便表示成功地切換了網路卡，此時原有的傳輸連線便自動的恢復。 附帶一提的是，圖中 NAPT 的部分不一定會存在，當它存在的時候便是遵照顯前所提及的 RFC 3519 來提供私有網際網路位址的支援，同時，註冊訊息與封裝封包在通過 NAPT 時會有 UDP 埠號轉換的發生。

以上便是由區域網路切換到無線區域網路時所發生的訊息交換流程。 而當威爾坑工程師結束會議回到他自己的辦公室時的情況又是如何，我們透過圖 4-4 便可瞭解。

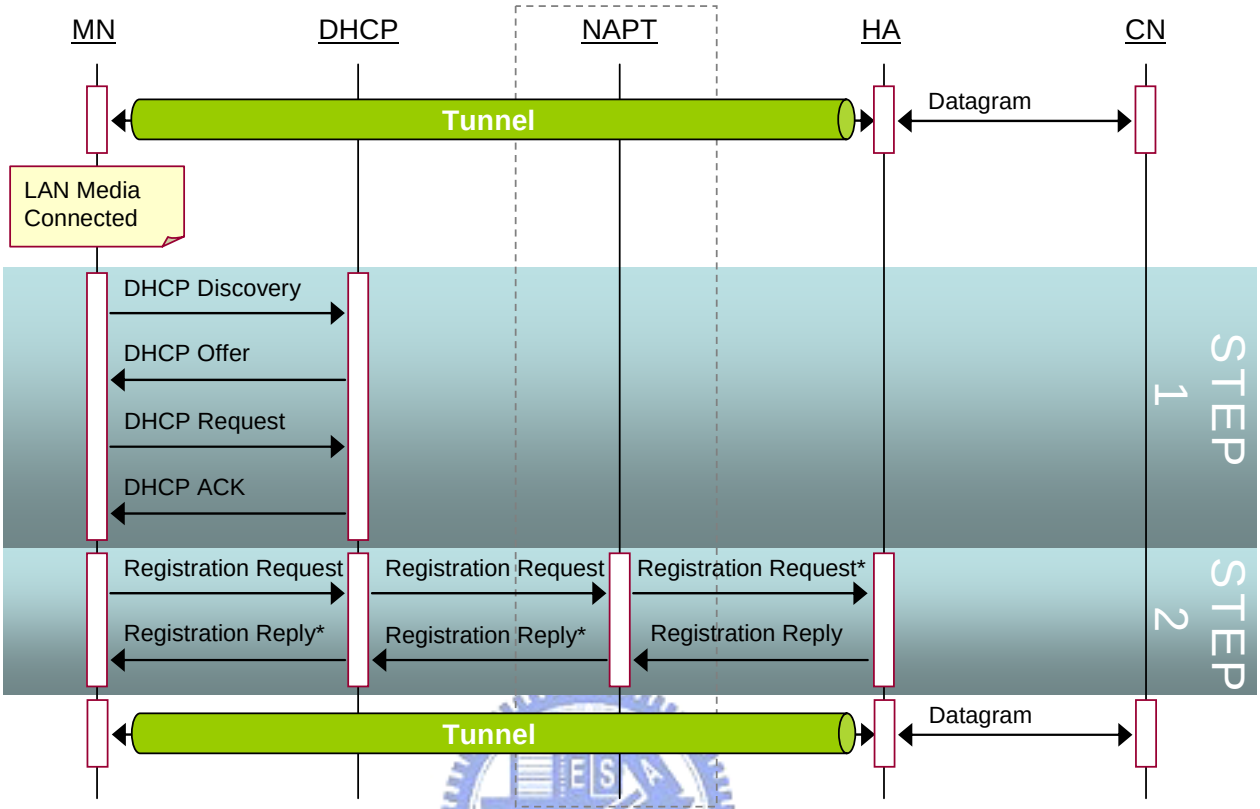

Figure 4 - 4: WLAN Handoff to LAN Message Flow

如同前述，當威爾坑工程師回到自己的座位，插上原有的網路線時，驅動程式所觸發的網路已連線訊息會使得 TCP/IP 協定堆疊內的 DHCP 子系統去重新取得 IP 位址。 當完成位址的取得，RIOMIP 程式便透過 Mobile IP 的註冊請求訊息嘗試切換回使用區域網路卡。 同樣的，在收到成功的註冊回覆訊息之後，原有的傳輸連線便不再經由無線區域網路來傳送而改走剛才插上的網路線。

4.3.2 WLAN-GPRS Roaming Scenario and Message Flow

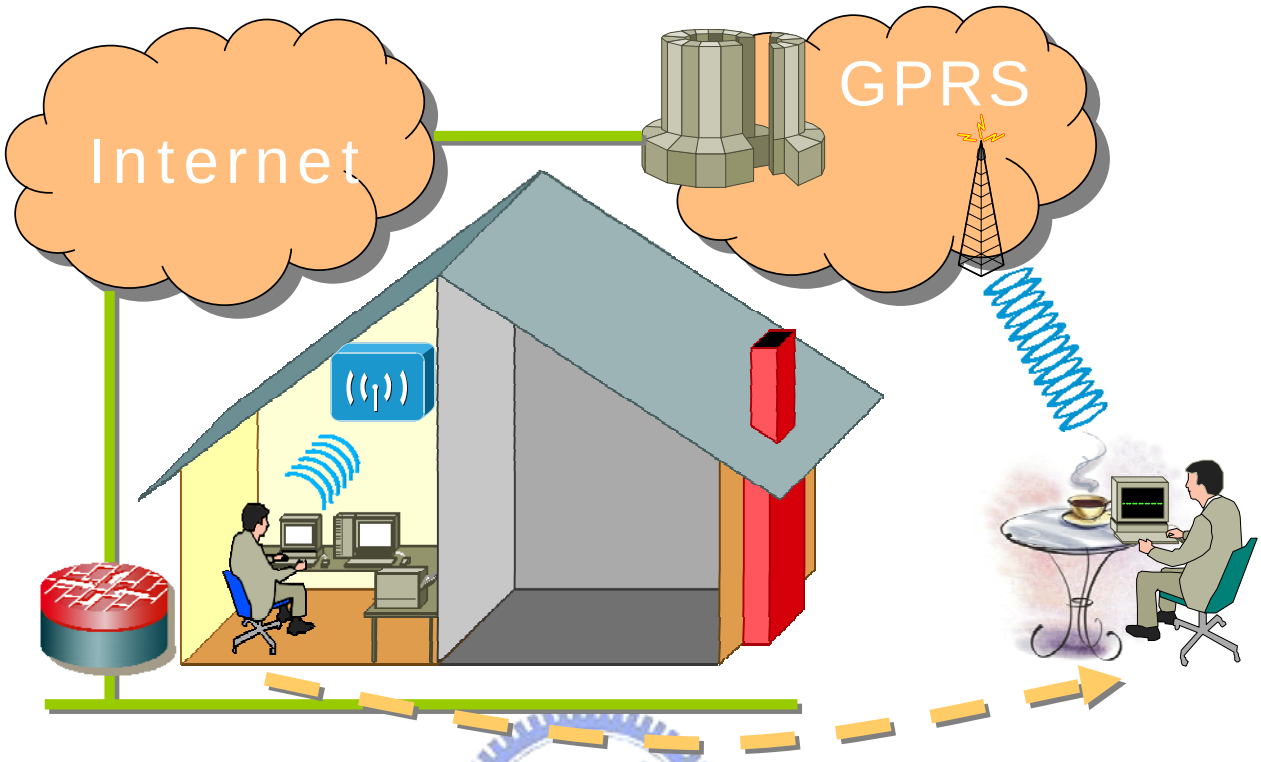

Figure 4 - 5: WLAN-GPRS Roaming Scenario

第二個實例(圖 4-5)是關於無線區域網路與整合封包無線電服務的切換。 如同前述，在亞瑟科技公司任職的威爾坑工程師感受到無線區域網路的便捷與迅速，決定請公司在他的辦公室內也安裝一台存取器，如此一來便不用每次開會都必須將網路線又拔又插的。 這天他接到遠在歐洲的好朋友透過著名的網際網路電話(Voice over Internet Protocol, VoIP)軟體 Skype 所打給他的電話。在接起電話聊了幾句後，他意識到這是一個長時間的通話而辦公室並不太適合這樣的行為。 所以他決定帶著升級過後插有 GPRS 數據卡的迅馳行動運算技術筆記型電腦到戶外的中庭與朋友好好聊一聊順便喝杯咖啡。 此時，透過 RIOMIP 智慧的網路切換機制，威爾坑先生便可走到哪講到哪，不會因為網路切換造成的連線中斷，讓原本享受科技與省錢的美意變成電信業者常掛在口中撥打國際電話改用 00X 的理由。

如同前面的實例，我們透過圖 4-6 也來檢視一下訊息的流程：

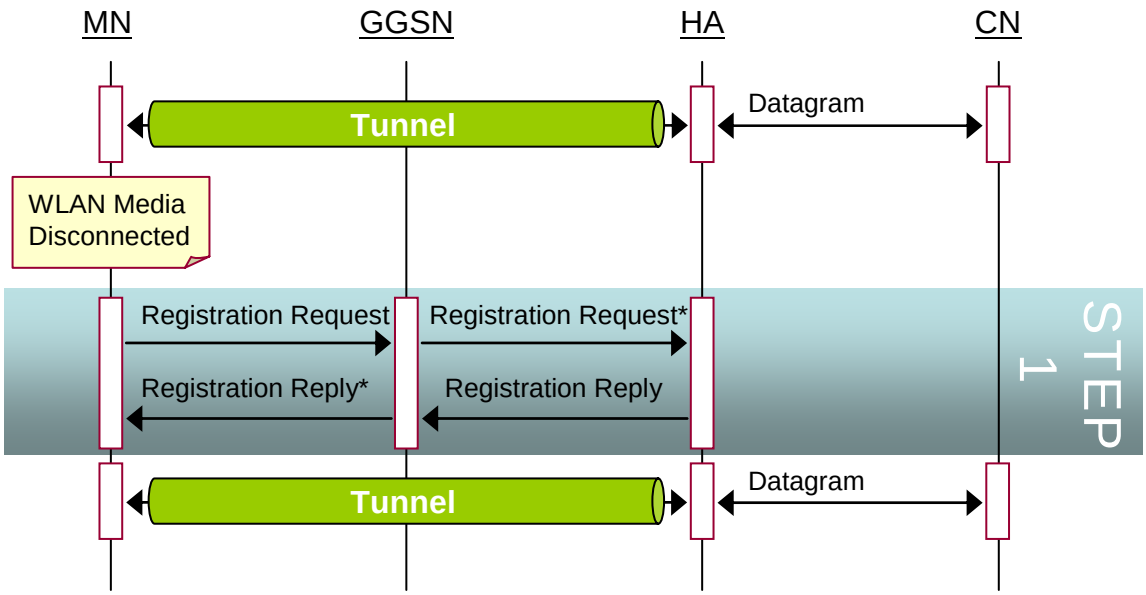

Figure 4 - 6: WLAN Handoff to GPRS Message Flow

在威爾坑工程師步行前往中庭的路上，當無線區域網路卡離開存取點所涵蓋的範圍，底層的驅動程式同樣因為失去與存取點的連結而觸發一網路電纜已拔除的訊息。 如圖 4-6 所示，當此訊息被 RIOMIP 所擷取時，便開始嘗試使用已建立連線的 GPRS 數據卡。 由於 GPRS、PHS 或是 CDMA 2000 等類型的數據卡，在連線時皆採用撥接之模式，換句話說，在撥接成功後便會取得連線用的 IP 位址等相關資訊。 此外，該類型的網路多半以傳送封包的數量來計費，因此在我們設定的情境之下該類型的數據卡是永遠保持連線的狀態。 所以，有別於前面的實例，在此無須等待網路連線的建立與等候透過 DHCP 來取得 IP 位址。 所必須的僅剩下透過該數據卡送出 Mobile IP 註冊請求此一步驟。 當 RIOMIP 收到成功的註冊回覆訊息後，便表示成功地切換到使用該數據卡，此時原有的傳輸連線便自動的恢復。

而當威爾坑工程師結束通話再次回到有存取點所涵蓋範圍的辦公室，RIOMIP 程式會再次地為使用者切換回無需付費的無線區域網路。 接下來我們便透過圖 4-7 來瞭解其切換過程中訊息的流程。

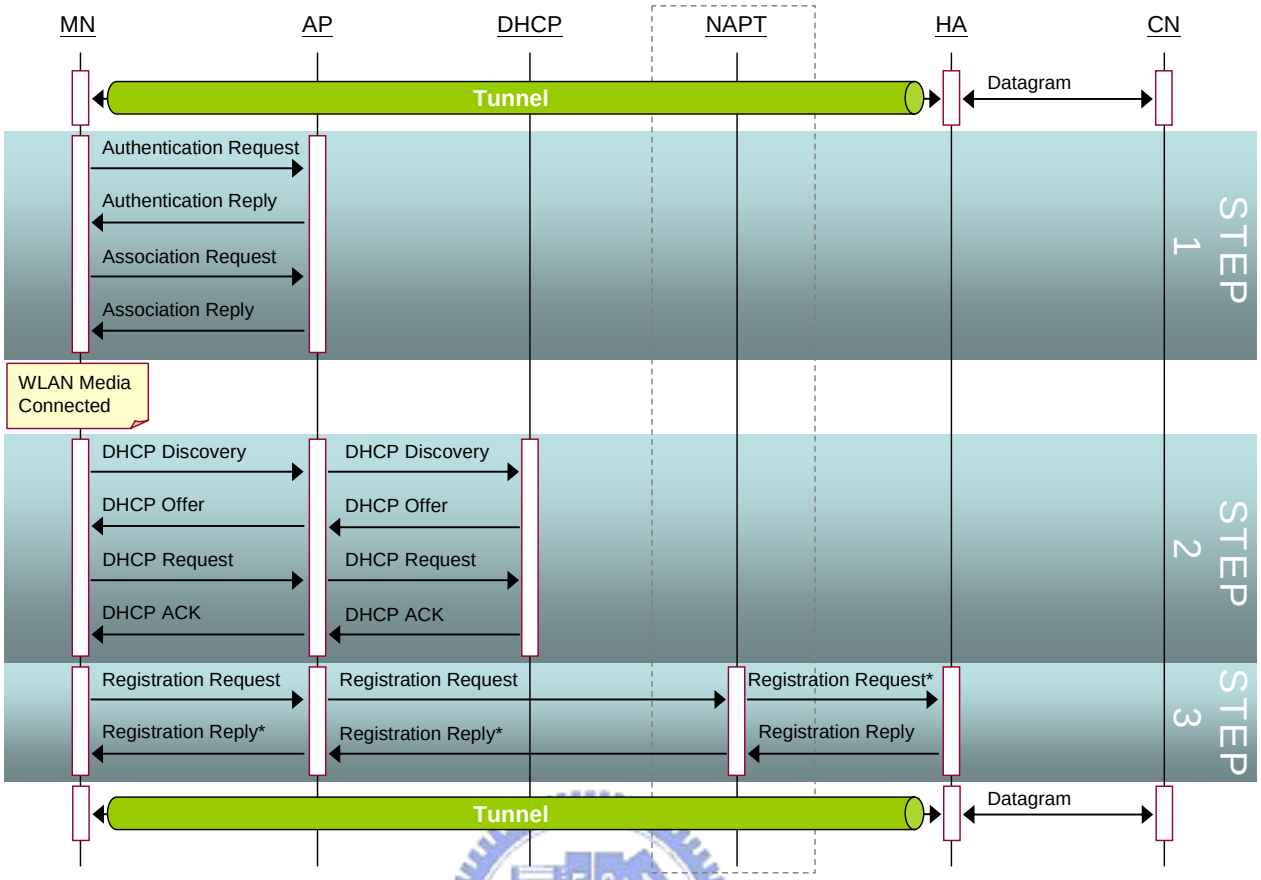

Figure 4 - 7: GPRS Handoff to WLAN Message Flow

首先，當無線區域網路卡偵測到其範圍內有可用的存取點時，底層的驅動程式便會自動地嘗試與之建立連結，也就是步驟一的行為。 當連結成功的被建立後，網路已連線的訊息同樣會觸發 DHCP 的運作去重新取得 IP 位址。 當步驟二完成的同時，RIOMIP 的 DHCP 監控元件會將所取得之 IP 位址等相關資訊傳遞給負責 Mobile IP 的元件做資訊的更新並送出新的註冊請求訊息。 在 RIOMIP 收到成功的註冊回覆訊息後也表示步驟三的完成。 換句話說，原本使用 GPRS 傳輸的連線此時便自動地切換成改走無線區域網路。

Section 4.4 Software Components

在本節當中將針對行動端上所運行的軟體套件 RIOMIP，對其內的各個元件做一詳盡的功能介紹與說明。 而對於行動代理器上的軟體套件 Dynamics，由於我們只是純粹使用並無在其上做大幅度的修改，所以在此便省略該部分的解說，詳細的資料可參考 [33]。

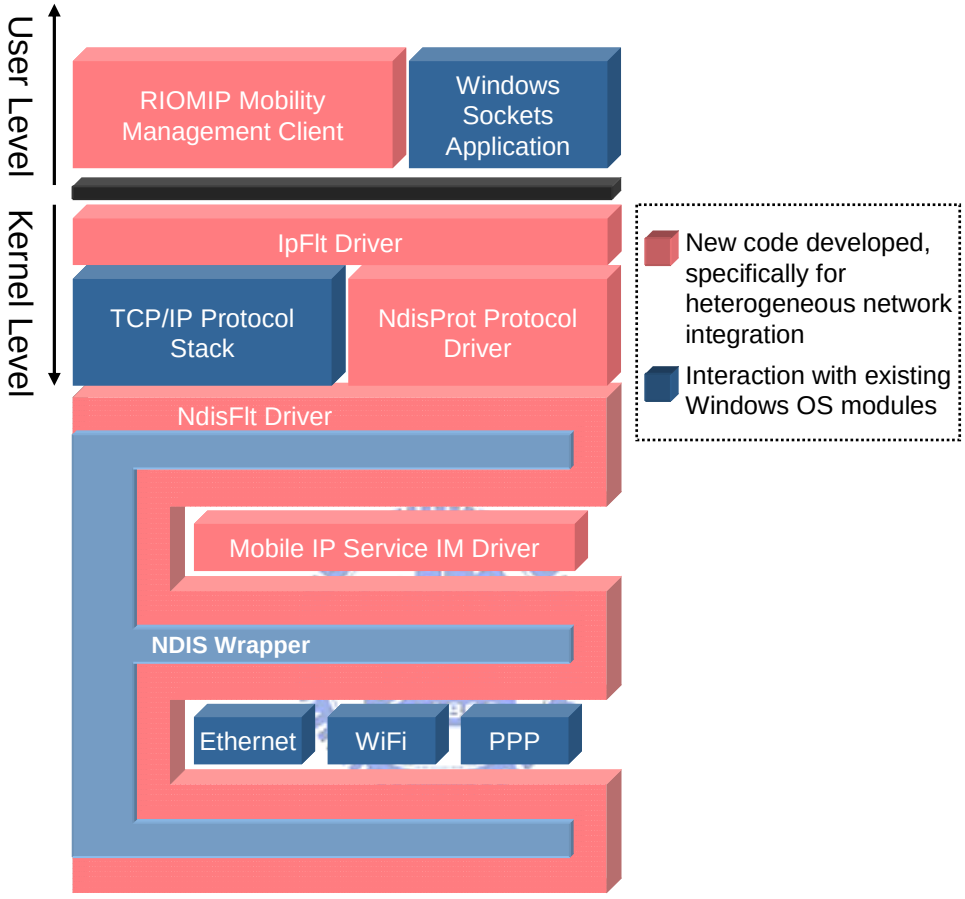

Figure 4 - 8: RIOMIP Client Software Architecture

圖 4-8 所展示的便是 RIOMIP 客戶端的軟體系統架構。 圖中深(藍)色的部分為視窗作業系統中原有的元件，包括在核心層的 TCP/IP 協定堆疊與各類型網路卡的迷你連接埠驅動程式，其中標示為 PPP 的裝置則是屬於 GPRS、PHS 與 CDMA 2000 等類型之數據卡所使用。 此外，在用戶層則有使用 Windows Sockets 所開發的網路相關應用程式。 上述的元件在整合的過程中皆無須有所變更或修改。 而淺(粉紅)色的部分則是為了整合異質網路所新增開發的程式，與 NDIS 部分相關的有 NdisProt 協定驅動程式，Mobile IP 服務中間層驅動程式以及 NdisFlt 驅動程式。 此外還有搭配 NdisFlt 所使用的 IpFlt 驅動程式。 以上部分皆屬於核心層之範圍。 唯一在用戶層的只有 RIOMIP 行動管理客戶端程式。

在接下來的小節內，將依序詳述方才所提及的各個新增元件，分別是 4.4.1 的 NdisFlt 驅動程式，4.4.2 的 IpFlt 驅動程式，4.4.3 的 Mobile IP 服務中間層驅動程式，4.4.4 的 NdisProt 協定驅動程式以及 4.4.5 的 RIOMIP 行動管理客戶端程式。

4.4.1 NdisFlt Driver

NdisFlt 驅動程式事實上就是在 2.3.3 中所提及的 NDIS 攔截驅動程式。 它主要的功能是提供兩個應用程式介面，其一為用來置換原有的 NDIS 應用程式介面以提供基本的 NDIS 驅動程式來呼叫，另一則用來與真正執行封包過濾的核心模式封包篩選引擎溝通。

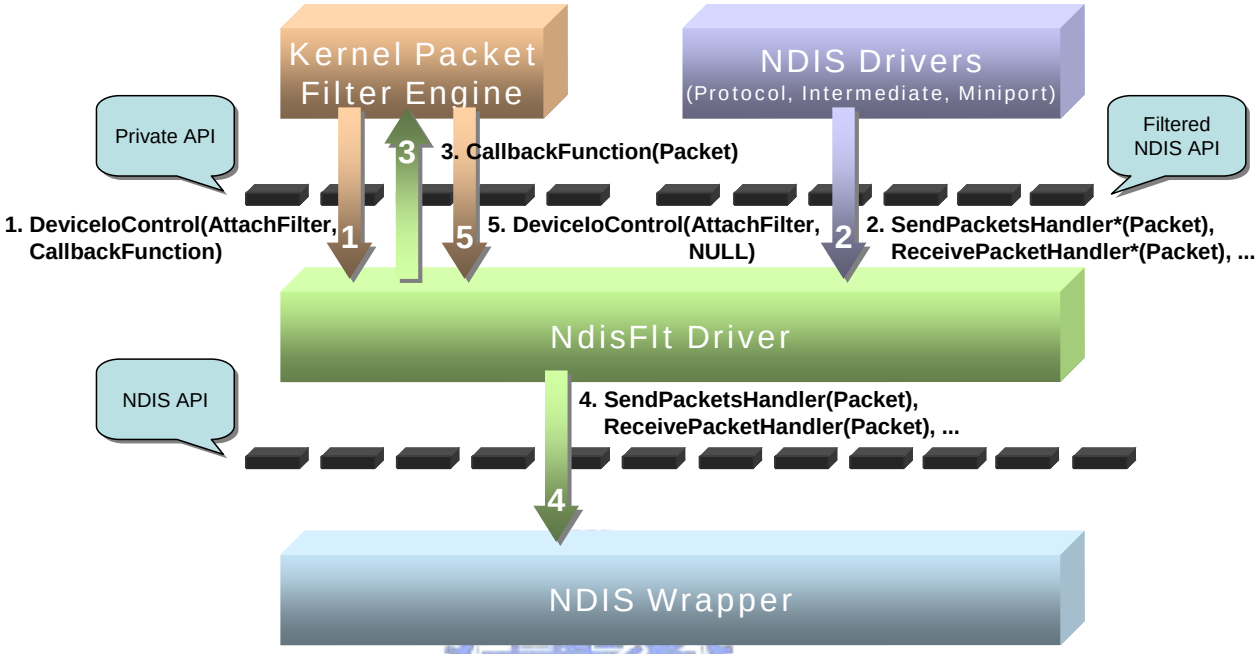

Figure 4 - 9: NdisFlt Driver Working Flowchart

透過圖 4-9 的運作流程圖便可以更清楚的瞭解 NdisFlt 驅動程式所扮演的角色，接下來便詳細說明每一個步驟：

1. 核心模式封包篩選引擎為一針對封包標頭或內容進行過濾與擷取動作的驅動程式。 它必須藉由 NdisFlt 驅動程式所提供的私有應用程式介面來運作。 首先，如同其它核心驅動程式間的溝通方式，核心模式封包篩選引擎利用 DeviceIoControl 命令將其內用來過濾封包的回呼函式告知 NdisFlt 驅動程式，也就是完成篩選器的裝設動作。
2. 當 NdisFlt 驅動程式載入系統後，它會置換掉原本用來傳送與接收封包的函式指標，如 SendPacketsHandler、ReceivePacketHandler 等。 因此在 NDIS 驅動程式(包括協定驅動程式，中間層驅動程式與迷你連接埠驅動程式)透過 NDIS 應用程式介面呼叫這些函示時，事實上被呼叫到的函示是由 NdisFlt 驅動程式所提供的。
3. 在 NdisFlt 驅動程式提供的取代函式內，會呼叫步驟一的封包過濾回呼函式並且把包含完整標頭與內容的封包當作該函式的參數傳遞給核心模式封包篩選引擎。
4. 如果封包篩選引擎決定讓封包通過，則 NdisFlt 驅動程式便呼叫相對應的原始 NDIS 應用程式介面所提供的函式處理該封包，否則該封包將不被繼續處理，也就是立刻遭到丟棄。

5. 最後，當核心模式封包篩選引擎結束其工作，它必須透過與步驟一相同的命令(除了在回呼函式填上 NULL)來移除封包篩選器。

4.4.2 IpFlt Driver

IpFlt 驅動程式實際上就是一種 4.4.1 所提到的核心模式封包篩選引擎。 它主要的功能除了上述的過濾封包以外，還負責將所擷取的封包複製一份給在用戶模式的封包監視程式(Packet Sniff Program)。

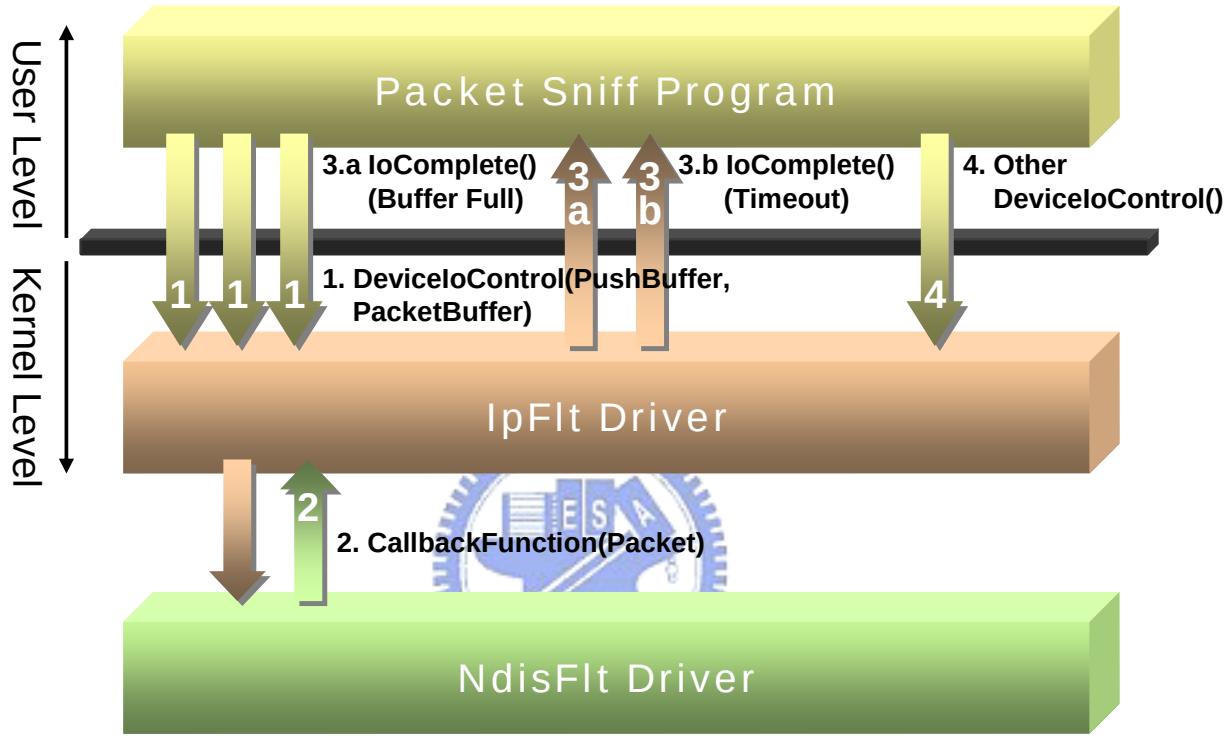

Figure 4 - 10: IpFlt Driver Working Flowchart

透過圖 4-10 我們便來瞭解一下 IpFlt 驅動程式運作流程的步驟：

1. 由於一般用戶模式下的應用程式無法將過濾封包的回呼函式直接傳遞給在核心模式的 NdisFlt 驅動程式(記憶體空間不同)，所以要在用戶模式下取得正在傳送或接收的封包就必須透過 IpFlt 驅動程式來完成。 首先，與任何驅動程式溝通都必須透過 DeviceIoControl 命令，因此用戶模式的封包監視程式就以該命令傳遞用來存放封包的記憶體空間供 IpFlt 驅動程式使用。 這裡必須特別注意，此處所使用的 DeviceIoControl 命令是採用非同步的方式。 換句話說，在該命令在下達後會立即得到正在處理中的回覆訊息，也就是該命令在稍後才會真正的完成。 這樣的目的是可以讓封包監視程式一次傳遞多組記憶體空間給 IpFlt 驅動程式以處理不及造成避免封包的遺漏。

2. 如同 4.4.1 的步驟三，當 IpFlt 驅動程式的過濾封包回呼函式被 NdisFlt 驅動程式所呼叫時，IpFlt 驅動程式除了判斷是否該丟棄該封包以外，還會將完整的封包內容填入步驟一的記憶體空間內。
3. 而在步驟一所下達的 DeviceIoControl 命令會在兩種情況之下被完成，首先是當該命令內的記憶體空間被填滿時(3.a)，再者是當固定的週期時間到達(3.b)。 當這兩者其中的一項事件發生時，封包監視程式便可從完成的 DeviceIoControl 命令內得到目前正在傳送或接收的封包。 固定週期時間的設計是為了反應即時的封包傳遞狀況，目前的設定為每 5 百萬分之一秒(microscend)。
4. 除了上述主要的 DeviceIoControl 命令外，IpFlt 驅動程式還提供其它用來控制封包過濾的相關 DeviceIoControl 命令。 例如設定過濾特定 IP 位址，設定過濾 ARP 請求與回覆訊息等。

4.4.3 Mobile IP Service Intermediate Driver

在 Windows 2000 之後版本的微軟視窗作業系統具備了媒體感應(Media Sense)的功能，其用來偵測網路媒體是否處於連結狀態。 連結狀態是指連線或插入網路上的實體媒體。 例如當你將網路線拔離網路卡時，在工具列會立即跳出一個寫著網路電纜已拔除的訊息視窗，而在你接回網路線時，也會有個視窗告知你區域網路已連線，如圖 4-11 所示。

Figure 4 - 11: Windows Media Sense

媒體感應這項功能是在 NDIS 版本 5.0 之後所加入的，TCP/IP 協定堆疊利用這樣的連線指示來協助網路的自動設定。 也就是說，媒體感應事件會觸發 DHCP 用戶端採取行動，例如試圖取得租用權 (發生連線事件時)，或者使介面和路由失效 (發生中斷連線事件時)。 因此，當網路電纜已拔除的訊息出現時，NDIS 會移除迷你連接埠驅動程式與協定驅動程式間的繫結(Binding)。 取消繫結所造成的是原有網路卡所使用的 IP 位址與其相關的路由被移除。 然而，在我們的實作當中這樣的行為同樣會造成切換網路卡時的連線中斷(詳細的原因將在第六章中說明)。 因此我們便設計了 Mobile IP 服務中間層驅動程式來解決這樣的問題。 其詳細的運作流程與步驟如圖 4-12 所示：

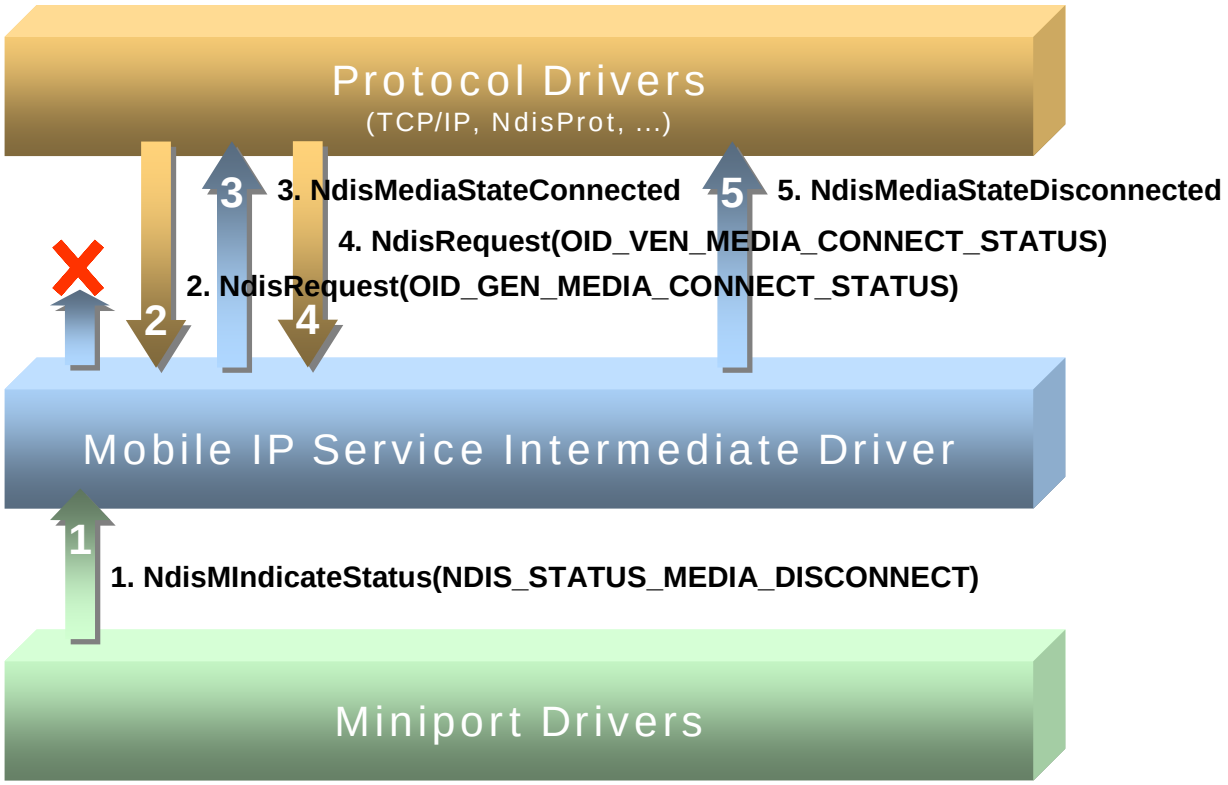

Figure 4 - 12: Mobile IP Service Intermediate Driver Working Flowchart

1. 當迷你連接埠驅動程式偵測不到實體媒體時，它會用 NDIS 所提供的應用程式介面內之函式 NdisMIndicateStatus 發送 NDIS_STATUS_MEDIA_DISCONNECT 之訊息告知上層驅動程式發生中斷連線事件。 而由先前所提及的網路層級化之架構可瞭解，該訊息首先會由中間層驅動程式所處理，也就是我們所開發的 Mobile IP 服務中間層驅動程式。 當它接收此訊息後，會更新其內部用來記錄連結狀態的變數，但是不再向上傳遞該訊息。 如此一來，原有的上層協定驅動程式，諸如 TCP/IP 協定驅動程式以及稍後會提到的 NdisProt 協定驅動程式等等，便無法得知媒體感應事件的發生。 如此一來，協定驅動程式會認為底層的實體媒體永遠存在，因此它與迷你連接埠驅動程式間的繫結便不會被移除。
2. 協定驅動程式除了利用 NdisMIndicateStatus 函式來得知網路是否處於連結狀態外，還可以使用同樣由 NDIS 所提供的 NdisRequest 函式來取得迷你連接埠驅動程式內用來記錄目前連結狀態的物件識別元 OID_GEN_MEDIA_CONNECT_STATUS。 因此我們的 Mobile IP 服務中間層驅動程式也必須處理這樣的情況。
3. 當協定驅動程式發出步驟二所提到的 NdisRequest 命令後，同樣由於網路層級化架構的關係，該命令回先被 Mobile IP 服務中間層驅動程式所處理。 此時不管網路是否為連結狀態，它都會直接回覆 NdisMediaStateConnected 告知協定驅動程式目前網路媒體是處於連結狀態。

4. 由以上兩個步驟的處理過程敘述可知道，協定驅動程式不管是用步驟一的 NdisMIndicateStatus 或是步驟二的 NdisRequest，皆無法得知目前底層網路真正的連結狀態。 雖然如此一來達到了我們所預期的功能，但是為了取得網路確實的連結狀態以供稍後會提到的行動管理客戶端程式作網路切換的判斷，所以 Mobile IP 服務中間層驅動程式還必須提供額外的方法來取得網路媒體是否處於連結狀態。 該方法如同步驟二所提到的下達 NdisRequest 命令，但是此時並非要求原本的物件識別元 OID_GEN_MEDIA_CONNECT_STATUS，而是由我們所自訂的物件識別元 OID_VEN_MEDIA_CONNECT_STATUS。
5. Mobile IP 服務中間層驅動程式在接收到上層協定驅動程式欲取得物件識別元 OID_VEN_MEDIA_CONNECT_STATUS 的要求後，會根據步驟一所提到內部用來記錄目前連結狀態的變數之值(NdisMediaStateConnected 或 NdisMediaStateDisconnected)回覆發送要求的協定驅動程式。

4.4.4 NdisProt Protocol Driver

NdisProt 驅動程式屬於在 2.3.1 中所提及的協定驅動程式，但它的作用並非真正的實作某種特定的傳輸協定堆疊，而是作為用戶模式程式與核心模式 NDIS 驅動程式間溝通的橋樑。 也就是說，NdisProt 協定驅動程式只提供私有的應用程式介面而沒有標準的傳輸層驅動程式介面。 一般說來，存在迷你連接埠驅動程式內的物件識別元(Object Identifier, OID)只能由在其上的其他 NDIS 驅動程式來存取。 這些物件識別元所記錄的是關於網路卡運作特性與統計資料的資訊，其命名皆以 OID 開頭，例如前面所提到的網路媒體連結狀態 (OID_GEN_MEDIA_CONNECT_STATUS)，或是存放乙太網路卡目前使用之卡號 (OID_802_3_CURRENT_ADDRESS)以及封包傳送成功之統計 (OID_GEN_XMIT_OK)等等。 因此，倘若用戶模式的應用程式對該資訊有興趣，則它就必須透過 NdisProt 此類的協定驅動程式或是中間層驅動程式來取得。 同樣的道理，先前所提及的迷你連接埠驅動程式運用 NDIS 函式 NdisMIndicateStatus 通知上層驅動程式事件的發生，例如網路媒體的連線/中斷 (NDIS_STATUS_MEDIA_CONNECT / NDIS_STATUS_MEDIA_DISCONNECT)或是連線速度的改變 (NDIS_STATUS_LINK_SPEED_CHANGE)等，也必須透過 NdisProt 協定驅動程式或是中間層驅動程式才能讓用戶模式的應用程式分享此訊息。 除此之外，NdisProt 協定驅動程式還提供用戶模式的應用程式直接對網路卡之迷你連接埠驅動程式下達封包傳送命令的介面。 如此一來，該封包無須經過系統原有的協定堆疊處理 (如 TCP/IP 協定堆疊內的路由與實體位址解譯)就可直接送出。 雖然這樣可以省去原本協定堆疊所需要的封包處理時間，但相對的，原本協定堆疊的工作就必須被下達傳送命令的應用程式以其他方法所取代。 而不管其使用何種的方式，迷你連接埠驅動程式接收到的必須是一個完整的封包，以乙太網路為例，完整的封包是指包含來源與目的 MAC 位址的完整乙太訊框。

圖 4-13 便是 NdisProt 協定驅動程式詳細的運作流程示意圖：

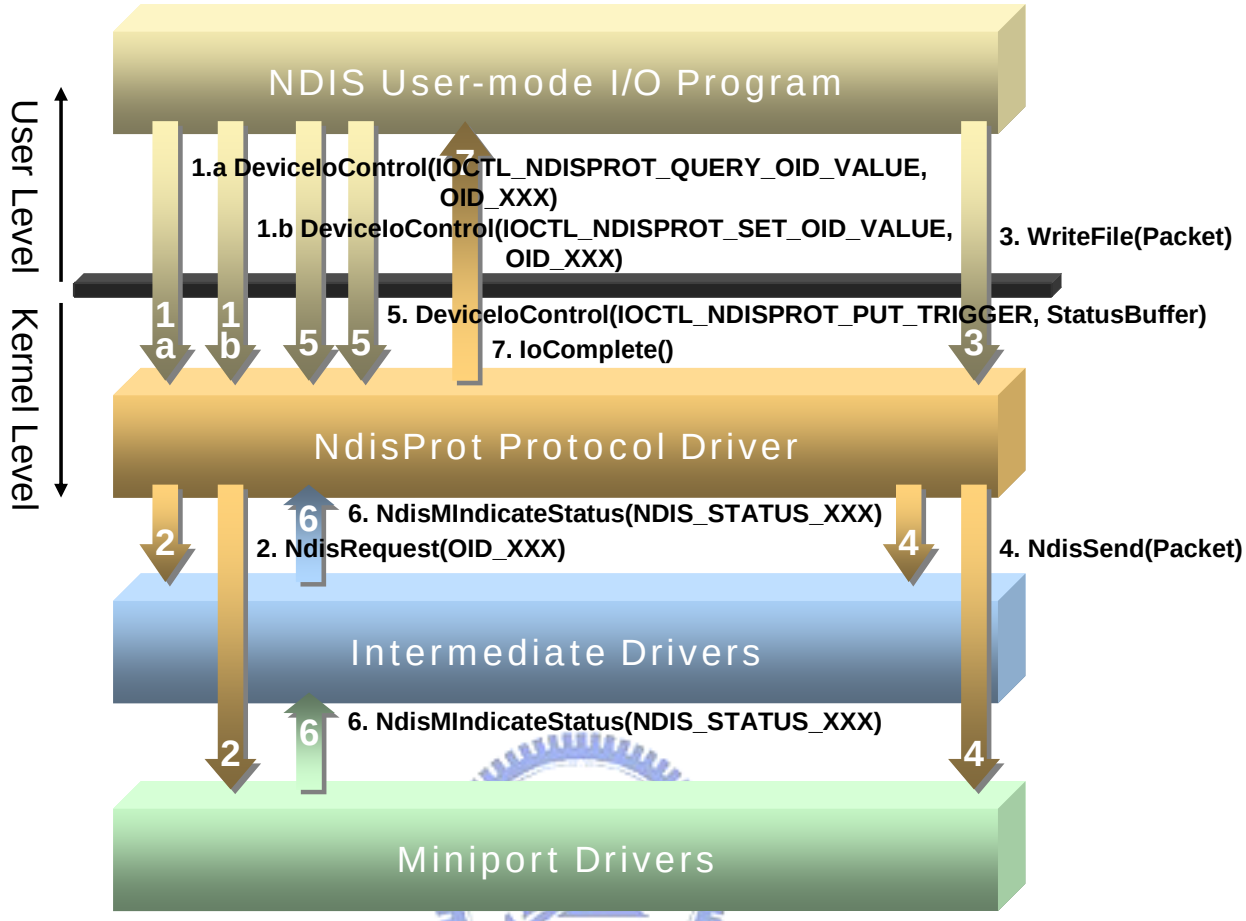

Figure 4 - 13: NdisProt Protocol Driver Working Flowchart

1. 欲存取 NDIS 資源的用戶模式應用程式一律透過標準的 DeviceIoControl 命令與 NdisProt 協定驅動程式進行溝通。 在參數方面，分別可以用 IOCTL_NDISPROT_QUERY_OID_VALUE 與 IOCTL_NDISPROT_SET_OID_VALUE 來取得或是設定迷你連接埠驅動程式內的物件識別元。
2. 在 NdisProt 協定驅動程式接收到步驟一的控制命令後，會將對該物件識別元的要求轉換成對應的 NdisRequest 函式後往下層的驅動程式如中間層驅動程式(如果存在的話)或是迷你連接埠驅動程式發送。
3. 而當用戶模式應用程式欲直接傳送封包時，則透過 WriteFile 命令將該封包傳遞給協定驅動程式。
4. 如同步驟二的動作，NdisProt 協定驅動程式會將步驟三的封包重新以 NdisSend 函式包裝送往下層的驅動程式如中間層驅動程式(如果存在的話)或是迷你連接埠驅動程式來完成後續的處理。
5. 而在接收迷你連接埠驅動程式的事件通知訊息方面，則是採用一種稱為逆向呼叫(Inverted Call)的模式 [30]。 首先先由用戶模式應用程式送出一個或多個帶有已配置記憶體空間用來

存放事件狀態的 DeviceIoControl 命令，其搭配的參數為 IOCTL_NDISPROT_PUT_TRIGGER。 而該命令不會立刻被 NdisProt 協定驅動程式所處理完成，因會它必須等到真正有事件發生的通知，因此取而代之的是它會先回覆命令處理中之訊息。

6. 當網路媒體狀態改變的事件由中間層驅動程式(如果存在的話)或是迷你連接埠驅動程式以 NdisMIndicateStatus 函式傳遞到 NdisProt 協定驅動程式時，它會任意選擇一個由步驟五所得到的 DeviceIoControl 命令，並將事件變數如 NDIS_STATUS_MEDIA_CONNECT 等存放在其內預先所配置的記憶體空間。
7. 最後以 IoComplete 函式完成步驟六所選定的 DeviceIoControl 命令。 如此一來，用戶模式的應用程式便可與其他的 NDIS 協定驅動程式一般即時的獲取來自迷你連接埠驅動程式關於媒體狀態改變的資訊。

4.4.5 RIOMIP Mobility Management Client

RIOMIP 行動管理客戶端程式是整個軟體套件的核心，它整合運用了上述各個小節內的核心驅動程式來達成異質網路間的漫遊服務。 其中包括網路配接卡的使用與管理，網路切換時的連線維持以及網路狀態之監控等等。 整個行動管理客戶端程式內所包含的軟體元件如圖 4-14 所示，共可分為六大模組。 以下便針對其內的各個軟體模組做逐一的介紹：

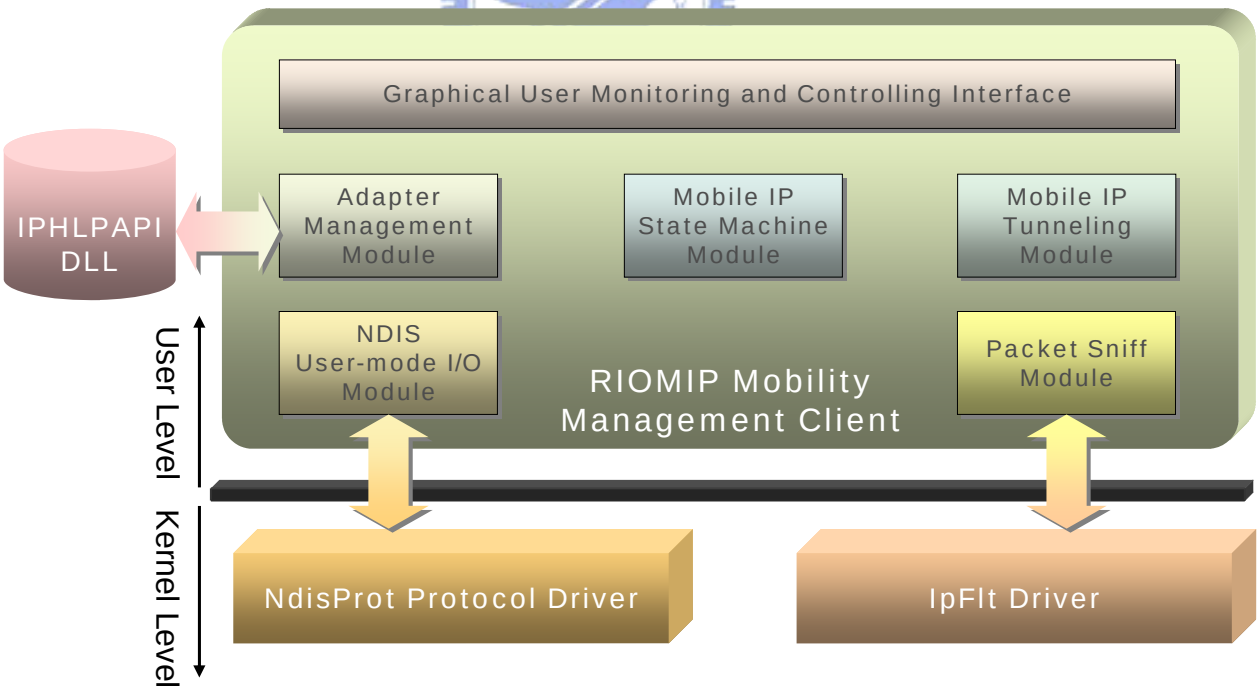

Figure 4 - 14: RIOMIP Mobility Management Client Software Components

- NDIS 用戶模式輸入/出模組 (NDIS User-mode I/O Module)

此模組的角色就同等於在 4.4.4 中所提到的用戶模式之應用程式，其負責與 NdisProt 協定驅動程式間的溝通。 主要的功能為直接往底層迷你連接埠驅動程式傳遞由其它模組所欲發送的封包以及自 NdisProt 協定驅動程式接收網路媒體事件通知並將該事件告知配接卡管理模組做適當的處理。 此外，其它的模組還可透過該模組取得迷你連接埠驅動程式內相關的物件識別元，例如乙太網路卡所使用之 MAC 位址或是無線區域網路卡目前之訊號強度等等。

- 封包監視模組 (Packet Sniff Module)

此模組即為 4.4.2 所提及的用戶模式封包監視程式，其透過 IpFlt 驅動程式監視與過濾所有來源或目的位址為家位址的 IP/ARP 封包(當行動端位在外地網域時)以及所有行動端所接收之 Mobile IP 相關訊息。 對於一般基於 Winsock 的網路應用程式所產生的封包，被封包監視模組所攔截後，將交由行動網際網路協定通道模組進行適當的封裝。 至於 Mobile IP 相關的封包，如果是經過封裝之封包，不管是透過 IP 承載 IP 或是 UDP 承載 IP 的方式，皆交給行動網際網路協定通道模組處理封包後續的拆裝動作。 而如果是 Mobile IP 之註冊訊息或是代理器公告則交給負責處理的行動網際網路協定狀態機模組。

- 配接卡管理模組 (Adapter Management Module)

對於網路卡的使用及切換便是此模組主要負責的項目。 在運作上與它交互配合的模組主要有三個，包括由系統所提供的 IPHLPAPI 動態連結函式庫(Dynamic Link Library, DLL)，配接卡管理模組可透過該函式庫存取及設定 TCP/IP 協定驅動程式內相關的參數，諸如網路卡的 IP 配置，路由表與 ARP 快取等。 其次為上述所提到的 NDIS 用戶模式輸入/出模組，透過該模組可得到網路卡驅動程式內相關之物件識別元以及網路媒體事件通知，可作為網路卡切換之判斷依據。 最後，當配接卡管理模組完成網路卡的切換動作後，它必須通知行動網際網路協定狀態機模組做後續相關的處理。 透過圖 4-15 可以更清楚地瞭解其運作的流程：

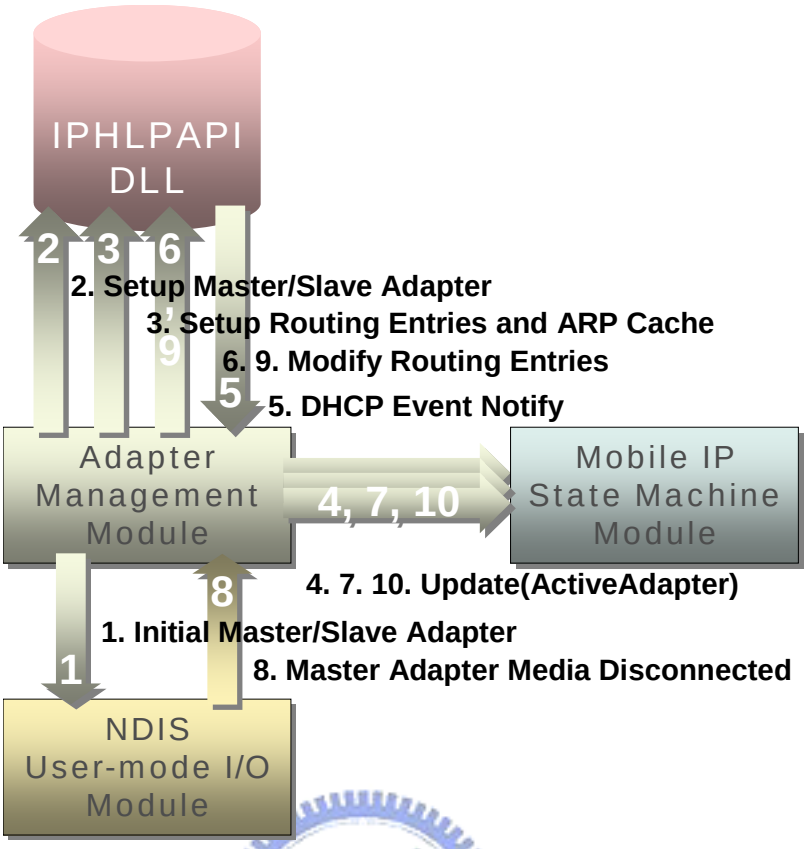

Figure 4 - 15: Adapter Management Module Working Flowchart

1. 在目前版本的 RIOMIP 行動管理客戶端程式設計上，可支援兩張網路卡的交替使用。 在實作上我們將這兩張網路卡以主配接卡(Master Adapter)與副配接卡(Slave Adapter)作為區別，而目前正在使用的配接卡則稱為運作配接卡(Active Adapter)。 當模組啟動後，它會對使用者所選定的主/副配接卡進行起始的動作，包括取得網路卡相關資訊如實體媒體型態與乙太網路之 MAC 位址等，以及完成取得底層媒體事件通知所需的準備工作。 以上動作皆透過 NDIS 用戶模式輸入/出模組來達成。
2. 接著透過 IPHLPAPI 動態連結函式庫所提供的應用程式介面來設定主/副配接卡的 IP 配置。 這部分主要的動作是將網路卡設定為自動取得 IP 位址，也就是透過 DHCP 協定來取得合適的 IP 位址作為配置轉交位址。 除此之外，最重要的步驟就是將行動端的家位址加在主配接卡的 IP 配置內，並設定主配接卡為運作配接卡。
3. 完成步驟二的網路卡 IP 配置後，同樣透過 IPHLPAPI 動態連結函式庫所提供的應用程式介面，配接卡管理模組將針對運作配接卡的 IP 配置以及 Mobile IP 協定的需求對 TCP/IP 協定堆疊內的路由表及 ARP 快取做適當的設定。 如此一來，一般基於 Winsock 的網路應用程式所產生的 IP 封包才皆會以行動端的家位址作為來源 IP 位址以符合 Mobile IP 協定的要求。

4. 模組啟動的最後一個步驟就是向行動網際網路協定狀態機模組送出更新運作配接卡的要求，在該要求送出後，行動網際網路協定狀態機模組將針對運作配接卡的配置轉交位址產生對應的註冊要求訊息發送給家代理器。
5. 利用 IPHLPAPI 動態連結函式庫所提供的監控 DHCP 協定應用程式介面可讓我們在系統重新取得 IP 位址時獲得一通知訊息。 該狀況可能發生在切換存取點時或是使用者以 ipconfig /renew 命令強迫更新 IP 位址。
6. 當模組收到該通知後會檢查新的 IP 位址與現有的配置轉交位址是否相同，若是已取得新的配置轉交位址則針對該位址以及 Mobile IP 協定的需求對 TCP/IP 協定堆疊內的路由表做適當的更動。
7. 如同步驟四的動作，向行動網際網路協定狀態機模組送出更新運作配接卡的要求。
8. 假設在模組啟動若干時間過後，接收到來自 NDIS 用戶模式輸入/出模組關於主配接卡媒體連結中斷的通知。 此時配接卡管理模組便開始進行網路切換的工作，首先它將檢查副配接卡是否準備妥當，包括其實體媒體是否已連結，有無合適的 IP 位址可作為配置轉交位址所使用等等。 若副配接卡已準備妥當則運作配接卡將換為副配接卡並繼續進行以下的步驟，否則此模組將持續的檢查主/副配接卡直到有可用的網路為止。
9. 如同步驟三的動作，模組將針對運作配接卡的 IP 配置以及 Mobile IP 協定的需求對 TCP/IP 協定堆疊內的路由表做適當的更動。
10. 最後，同樣向行動網際網路協定狀態機模組送出更新運作配接卡的要求完成網路卡的切換。

■ 行動網際網路協定通道模組 (Mobile IP Tunneling Module)

行動網際網路協定通道模組負責的是封包的封裝與拆裝動作，目前所支援的通道模式有傳統的 IP 承載 IP 以及為了穿越網路位址轉譯器所使用的 UDP 承載 IP 等兩種。 透過圖 4-16 可進一步瞭解封包的封裝與拆裝流程以及此模組與其它模組間的互動關係：

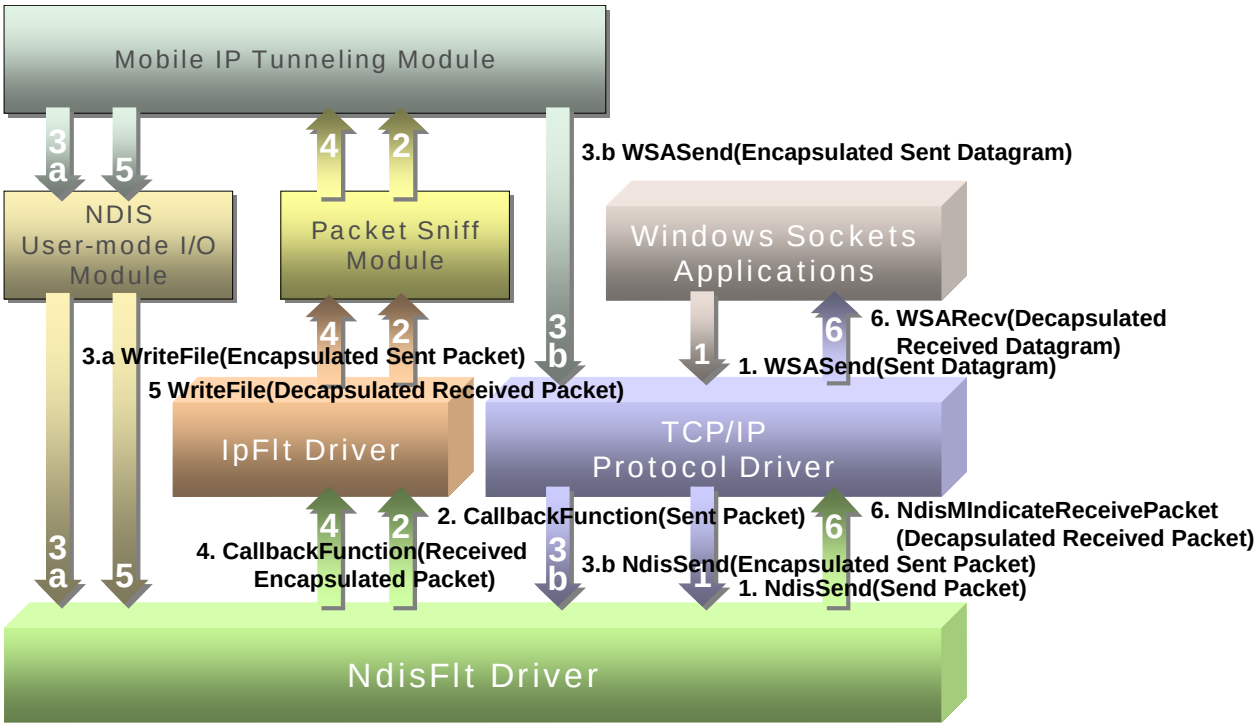

Figure 4 - 16: Mobile IP Tunneling Module Working Flowchart

1. 基於 Winsock 的網路應用程式所送出的 IP 數據資料經過 TCP/IP 協定堆疊的路由與實體位址解譯後所形成的完整封包，也就是一個乙太網路訊框，會被 NdisFlt 驅動程式所攔截。
2. IpFlt 驅動程式經由向 NdisFlt 驅動程式所註冊的回呼函式可得到該封包的拷貝。 經過檢查發現，該封包的來源位址為家位址，而行動端又位在外地網域，因此它會命令 NdisFlt 驅動程式丟棄該封包並將該封包的拷貝經由封包監視模組傳遞給行動網際網路協定通道模組進行必要的封裝。
3. 行動網際網路協定通道模組將視行動網際網路協定狀態機模組內對於通道模式的設定執行 IP 承載 IP 或是 UDP 承載 IP 的封裝動作。 在完成封包的封裝後，如果在 ARP 快取內已存有閘道器之 MAC 位址，則該模組便將經過封裝的 IP 數據資料加上適當的乙太網路訊框標頭並透過 NDIS 用戶模式輸入/出模組直接送出 (3.a)。 否則，該經過封裝的 IP 數據資料將循原本的途徑再次經由 TCP/IP 協定堆疊的路由與實體位址解譯後送出 (3.b)。 在此便可看出前者將擁有較好的效率。 而不管由何種途徑送出已封裝的 IP 數據資料，由於其來源位址皆已不是行動端的家位址，因此不會再由 IpFlt 驅動程式所攔截，而是直接由運作配接卡的迷你連接埠驅動程式所處理。
4. 而在接收經過由家代理所封裝的封包方面，如同步驟二的動作，IpFlt 驅動程式先命令 NdisFlt 驅動程式丟棄該封包後將該封包的拷貝經由封包監視模組傳遞給行動網際網路協定通道模組進行必要的拆裝。

5. 行動網際網路協定通道模組在移除適當的封包標頭後，會將已拆裝的 IP 數據資料加上來源與目的 MAC 位址皆為運作配接卡本身卡號的乙太網路訊框標頭並透過 NDIS 用戶模式輸入/出模組直接送出。
6. 步驟五所送出的乙太網路訊框會經由迷你連接埠驅動程式繞回 TCP/IP 協定堆疊作適當的處理(封包重組與 IP/UDP/TCP 標頭檢查碼的檢驗等)後，由相關的網路應用所接收。 不循步驟 3.b 直接送往 TCP/IP 協定堆疊的原因在於考量其內部設計的關係，此時封包重組的動作會被省略。 如此一來在實作上，行動網際網路協定通道模組就必須增加對於 IP 數據資料重組的處理，然而該動作並非相當容易。 因此在降低各個模組複雜度的考量上，還是交由系統的 TCP/IP 協定堆疊來執行該工作。

- 行動網際網路協定狀態機模組 (Mobile IP State Machine Module)

行動網際網路協定中的行動端程式即是在此模組中實現，它負責處理與發送所有行動網際網路協定相關的訊息，包括代理器公告與請求以及註冊要求與回覆等。 而內部的資料結構則儲存與行動網際網路協定相關的資料，例如目前可用之行動代理器與有效註冊等資訊。 除此之外，內部的狀態機維持著行動網際網路協定的正常運作。 模組內最主要也是最為重要的函式即為先前所提到的更新(Update)函式。 當該函式被呼叫後，此模組將視現階段內部狀態機所處的狀態以及運作配接卡目前的配置轉交位址決定下一個狀態以及所使用的通道模式為何，並在更新內部的資料結構後產生適當的註冊訊息送往家代理器。 其運作的細節部分將在稍後的第六章中詳述。

- 圖形化使用者監控介面 (Graphical User Monitoring and Controlling Interface)

此模組為一圖形化的使用者介面，讓使用者可以設定欲使用的主/副配接卡。 此外，透過其它模組的配合，即時的網路卡及行動網際網路協定相關資訊，例如網路卡媒體連結狀態，媒體訊號強度(如果可以取得)，配置轉交位址與通道模式設定等，皆會顯示在該介面上供使用者參考。 該介面還提供對行動網際網路協定狀態機模組下達執行更新函式的功能，讓使用者可以手動強迫切換所使用的配接卡。

Chapter 5 Alternative Tunneling for Mobile IP NAT Traversal

Section 5.1 Objectives

在 3.2.2 中所敘述的 IETF 解決方案成功地結合行動網際網路協定與網路位址轉譯器， 然而卻也造成了一些缺失。 首先，原始的行動網際網路協定是基於網路層來支援行動性。 但在套用 UDP 承載 IP 的封裝方式後，行動性的支援便上提到了傳輸層。 這不僅破壞了原有行動網際網路協定設計的標的，同時也造成了封包在通過網路位址轉譯器時需要更多額外的處理。 其次，在 RFC 3519 文件中更提及在該 UDP 通道被建立後，行動端可能需用經過 UDP 封裝的 ICMP 回應要求/回覆訊息作為維護訊息(Keepalive)，以維持在網路位址轉譯器上網路位址與埠號的對應關聯。 這意味著將產生一些傳統行動網際網路協定中所沒有的非必要流量，如果是在無線網路的環境，有限的頻寬很可能被這些維護訊息所佔去。 最後，我們都知道網路位址轉譯器是屬於全狀態(Stateful)的裝置，也就是說，它必須保有過去進出封包之網際網路位址與埠號的對應資訊。 基於這個原因，我們視網路位址轉譯器為單點故障(Single Point of Failure)。 換句話說，我們將無法以設定好與發生故障的位址埠號轉譯器有相同網際網路位址的備援機器來立即取代之。 在這情況下，由行動端所建立的 UDP 通道也將中斷。

有鑑於以上的種種因素，我們另行開發了一種讓行動網際網路協定穿越網路位址轉譯器的方法。 其設計的目標與準則如下：

- 節約公有網際網路位址的使用

本方法將基於網路位址埠號轉譯器的運作模式，也就是所有在私有網域內的主機將共享單一的公有網際網路位址來連線網際網路。

- 完全由網路層提供行動性的支援

在我們的設計之下，將使得行動網際網路協定對於私有網域內的行動性支援回歸到網路層。換句話說，在封包的封裝上，將不會使用任何如 TCP 或 UDP 等的傳輸層協定。 如此不管是在行動端，行動代理器或是網路位址轉譯器上的封包處理都將更加迅速。

- 最小化的額外訊息負擔

在封包的封裝上我們將改善 UDP 承載 IP 封裝模式所帶來的龐大額外新增標頭，並盡量回歸到傳統 IP 承載 IP 封裝模式的簡潔。 這樣的好處是可降低經過封裝的封包在傳輸的過程中遭遇多次的分割與重組並提升有效的頻寬使用率。 此外在我們的方法中也去除了維護訊息存在的必要性。 因此可預期的是傳輸速率的改善。

- 使網路位址轉譯器更趨無狀態(Stateless)化

由於本方法中行動性的支援純粹由網路層來提供，因此網際網路位址與埠號的對應關係便不復存在。 所以當支援本方法的網路位址轉譯器故障時便可以用備援的機器來立即取代之，而其下

所服務的行動端也無須做任何額外的動作來維持原有建立的通道，如此對於行動性的支援更增添了一層保障。

- 兼顧相容性的修改

無論在行動端，行動代理器或是網路位址轉譯器的修改上，我們的第一考量是保留其原有的運作，也就是我們所設計的方法皆以加值(Addon) 的模式插入原有的各個軟體元件之中。 意即我們的方法將具備向下相容的能力，當無法套用本方法時，將用原有的方式來提供網路位址轉譯器下的行動性支援。

Section 5.2 System Architecture

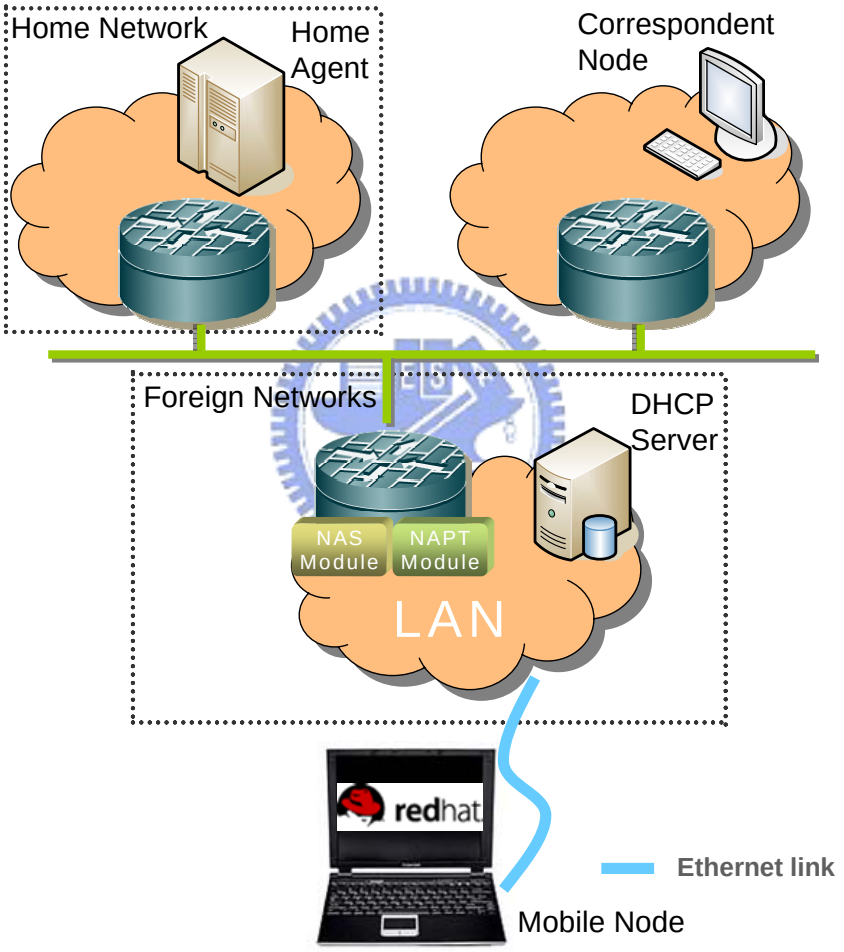

Figure 5 - 1: Mobile IP Alternative Tunneling System Architecture

圖 5-1 便是我們所開發的行動網際網路協定穿越網路位址轉譯器解決方案的系統架構示意圖。 由於我們的方法需要更動網路位址轉譯器，因此礙於現實的環境，僅能在我們所能掌控的有線或無線區域網路上實現。 在 Mobile IP 的運作上，我們同樣採用配置轉交位址模式，也就是說在外地網域下由 DHCP 伺服器負責提供轉交位址給行動端。 此外，該轉交位址為一私有網際網路位址。 當然，外地網域還有網路位址轉譯器負責對外的連線，該網路位址轉譯器為插有兩張乙太網路卡並運行 Linux 作業系統的伺服器，其上除了原有的網路位址埠號轉譯模組外，還搭載由我

們所設計的網路位址交換(Network Address Swapping, NAS)模組。 而在 Mobile IP 套件方面，無論是家代理器或是行動端，其上所使用的軟體皆修改自芬蘭赫爾辛基科技大學所開發的 Dynamics。

Section 5.3 Protocol Sketch

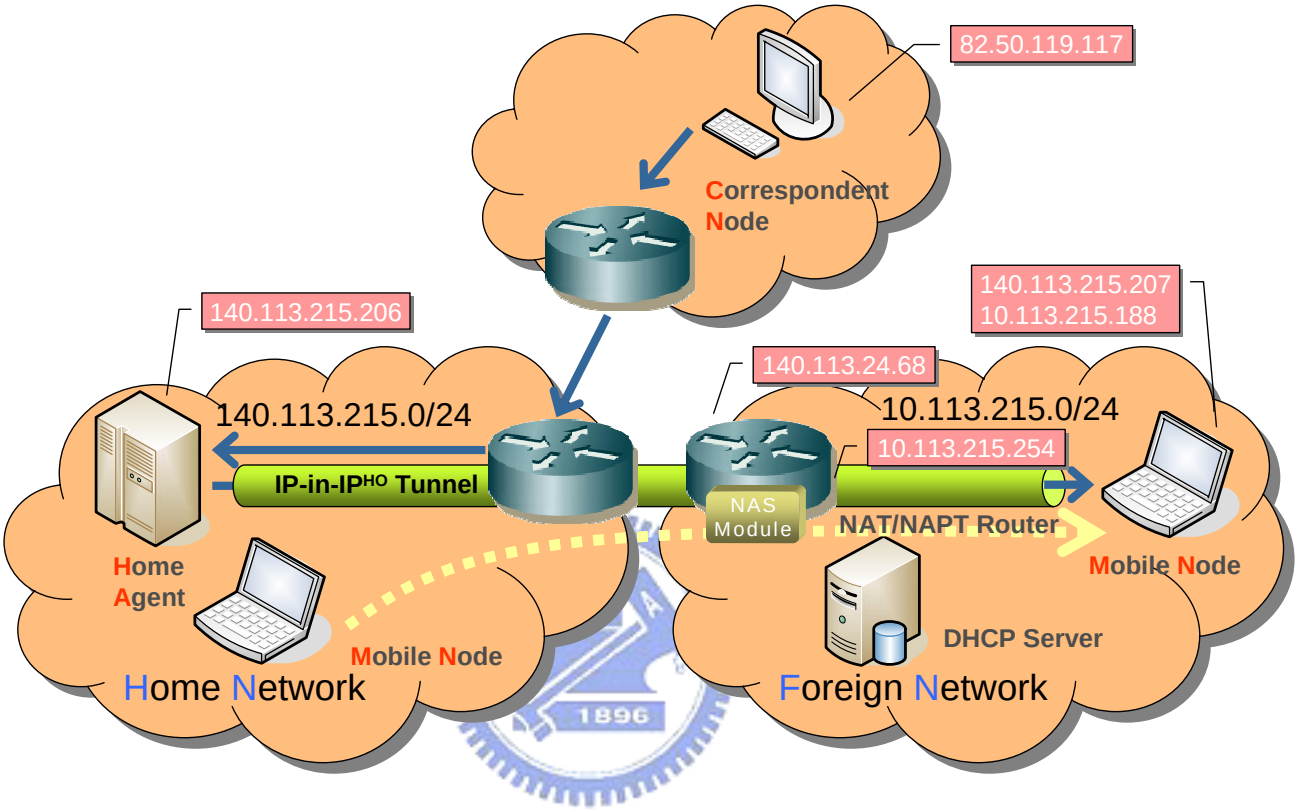

Figure 5 - 2: Alternative Mobile IP NAT Traversal Scheme

比較圖 5-2 與圖 3-15 的 IETF 解決方案，除了幾點相異的地方之外，我們可以發現它們相當的雷同。 首先是網路位址轉譯器的部分，基本上它還是屬於 2.2.2 中所提及的網路位址埠號轉譯器，不同之處在於其上搭載了稱為網路位址交換的模組。 該模組並不影響與改變原有網路位址埠號轉譯器對於通過封包的處理，只有當遇到特殊封包時才會介入。 換句話說，如果沒有特殊封包的出現，它就是一個普通的網路位址埠號轉譯器，這點便是先前所提及的兼顧相容性之修改。 而所謂的特殊封包是指帶有特定 IP 標頭選項(Header Option)的 IP 封包。 當該封包通過網路位址轉譯器時，網路位址交換模組會將該標頭選項內用來儲存 IP 位址的欄位與 IP 標頭內的來源或目的位址做交換，些許類似來源路由(Source Route)的動作。 由此可知，該模組的運作完全仰賴 IP 封包標頭內所提供的資訊而非網路位址轉譯器本身所維護的 IP 位址與埠號之對應關係，所以我們可以說該模組是無狀態的。 其次相異的部分則是家代理器與行動端之間所建立的通道，當雙方經由註冊的流程之中發現介於兩者之間的網路位址轉譯器上有運作中的網路位址交換模組，他們便會使用稱為 IP^{HO} 承載 IP 的封裝模式來取代原先的 UDP 承載 IP 封裝。 簡單的來說，該封裝模式便

是傳統 IP 承載 IP 封裝的變形，其過程是將行動端的轉交位址放置在前述特定的 IP 標頭選項內並附加在外層 IP 標頭之後。 因此，這樣的封裝便不涉及任何傳輸層的協定，也達到完全由網路層提供行動性支援的目標。

Section 5.4 Detail Design

在經過上述簡單的介紹過後，本節中將詳細地說明行動端如何與家代理器建立起 IP^{HO} 承載 IP 模式的封裝通道以及在網路位址轉譯器內的網路位址交換模組如何處理經過該封裝的封包。 首先 5.4.1 會介紹所我們所使用的 IP 標頭選項，其次 5.4.2 是網路位址交換模組的運作，最後在 5.4.3 則是完整的流程解說。

5.4.1 Data Structure

IP^{HO} 承載 IP 的封裝示意圖如圖 5-3 所示，事實上它正如同傳統的 IP 承載 IP 之封裝，只是在外層的 IP 標頭之後增加了一個標頭選項。 且該標頭選項的設計與使用皆遵守既有 RFC 791 文件的規範，也就是為一個型態-長度-數值(Type-Length-Value)的格式。 以下便一一介紹其內的各個欄位：

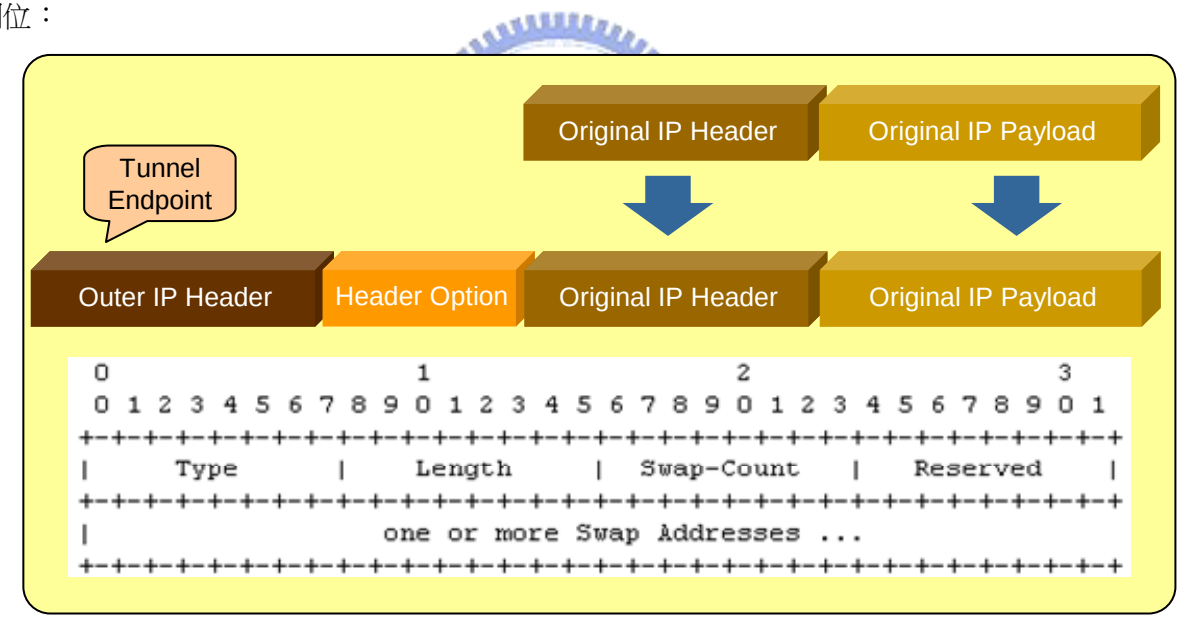

Figure 5 - 3: IP-in-IP^{HO} Encapsulation and Header Format

- 型態 (Type)

8 位元，暫訂的值為 252 (11111002)。 IP 標頭選項內的型態欄位格式如圖 5-4 所示，對照後可知複製欄位(C)為 1，表示此標頭選項必須複製到所有的封包分割片段內。 而類別欄位(Class)為 3 是為了避免影響現有的路由器對於封包的正確處理，該數值為網際網路協定在設計時所保留未使用的。 最後的選項(Option)欄位則為 28。

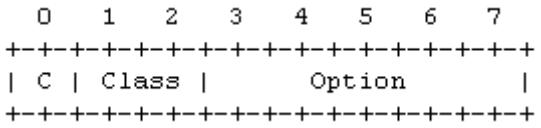

Figure 5 - 4: IP Header Option Type Field Format

- 長度 (Length)

8 位元，記錄此標頭選項的位元組長度。 假設在只放置了一個交換位址的情況下，此欄位的值為 8。

- 交換計數 (Swap-Count)

8 位元，記錄已執行的位址交換次數。 當網路位址轉譯器內的網路位址交換模組執行位址交換的動作後，必須將本欄位的值增加 1，而初始值應設為 0。

- 保留 (Reserved)

8 位元，未使用的欄位。 如欲附加本標頭選項應將之設為 0。

- 交換位址 (Swap Address)

可變動的長度，存放一個以上的網際網路位址，而每個網際網路位址為 32 位元。

5.4.2 Network Address Swapping (NAS) Function

網路位址轉譯器內的網路位址交換模組對於帶有型態 252 的 IP 標頭選項封包之處理依其流向可分為外送(Outbound)及內送(Inbound)兩個部分。 以下便針對這兩部分介紹網路位址交換模組的運作行為模式：

- 外送封包處理程序 (Outbound Packet Process)

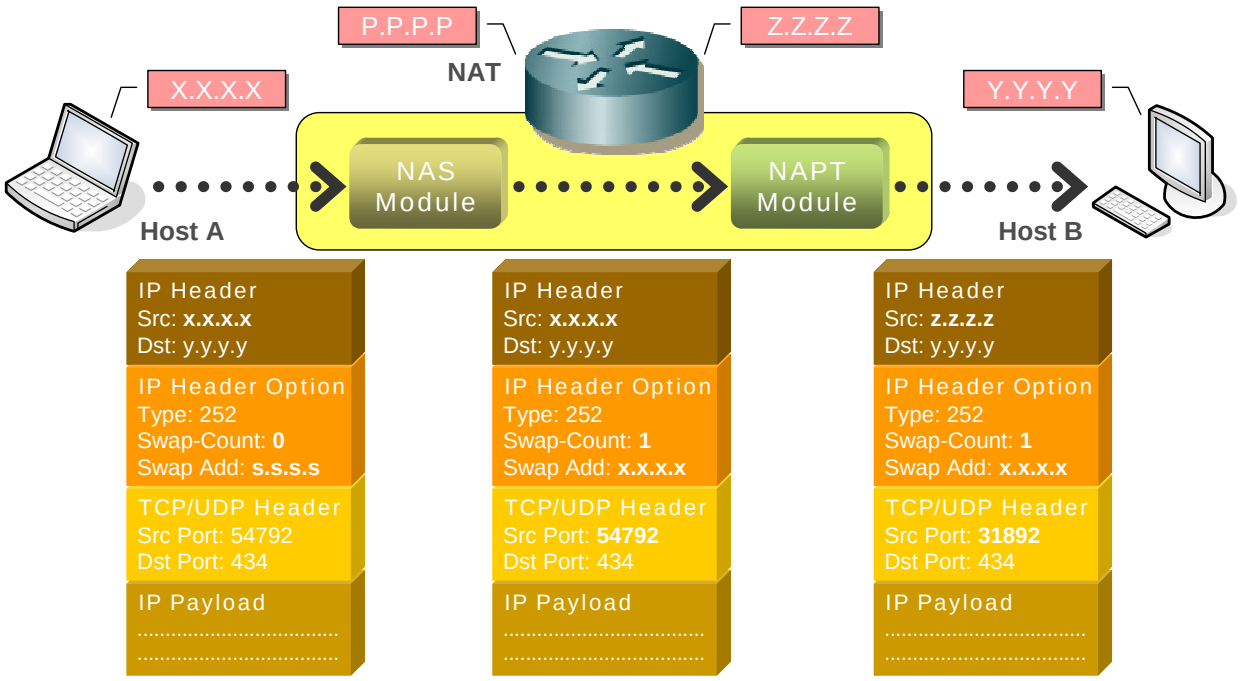

Figure 5 - 5: Network Address Swap Module Outbound Packet Process

當位在私有網域下的主機 A 發送一封包給在公有網域下的主機 B，如圖 5-5 所示，則該封包對於在兩者之間的網路位址轉譯器來說便是歸類於外送的封包。 而對於外送封包的處理程序則是先通過網路位址交換模組後再將處理過後之封包交給原始的網路位址埠號轉譯模組。 詳細的步驟如下：

1. 檢查 IP 封包是否帶有型態 252 之 IP 標頭選項，如果有則將封包交由網路位址交換模組處理，進入步驟二。 否則直接交給網路位址埠號轉譯模組，步驟三。
2. 網路位址交換模組複製 IP 標頭內的來源位址 x.x.x.x 至型態 252 之 IP 標頭選項內的交換位址欄位，並累加標頭選項內的交換計數欄位。 這裡必須特別注意的是，採用複製而非交換的動作是為了避免影響稍後網路位址埠號轉譯模組的運作。 因為該模組可能基於安全性的考量，只轉譯來源位址與其對內所使用的 IP 位址 p.p.p.p 同網域之封包。 倘若使用交換的動作，要是交換位址欄位內的 IP 位址 s.s.s.s 與 p.p.p.p 不同網域，則該封包便會在網路位址埠號轉譯模組內遭到丟棄。
3. 封包進入網路位址埠號轉譯模組，IP 標頭內的來源位址會被換成網路位址轉譯器對外的公有 IP 位址 z.z.z.z。 同時，傳輸層標頭(如果存在的話)內的來源埠號也會被置換。 至此便完成外送封包的處理程序。

- 內送封包處理程序 (Inbound Packet Process)

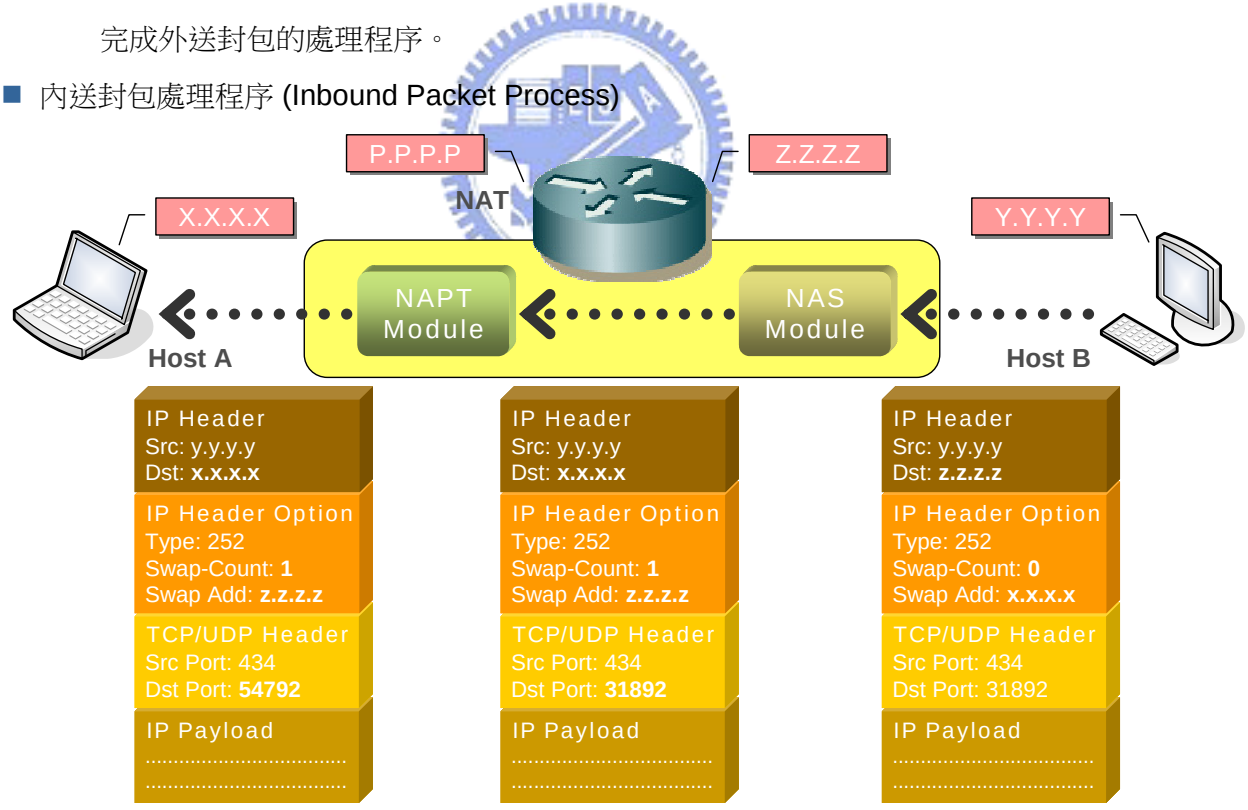

Figure 5 - 6: Network Address Swap Module Inbound Packet Process

而當在公有網域下的主機 B 回覆一封包給在私有網域下的主機 A，如圖 5-6 所示，則該封包對於在兩者之間的網路位址轉譯器來說便是歸類於內送的封包。 而對於內送封包的處理程序則如

同外送封包的處理，一樣先通過網路位址交換模組後再將處理過後之封包交給原始的網路位址埠號轉譯模組。 詳細的步驟如下：

1. 檢查 IP 封包是否帶有型態 252 之 IP 標頭選項，如果有則將封包交由網路位址交換模組處理，進入步驟二。 否則直接交給網路位址埠號轉譯模組，步驟三。
2. 網路位址交換模組交換 IP 標頭內的來源位址 z.z.z.z 與型態 252 之 IP 標頭選項中交換位址欄位內的 IP 位址 x.x.x.x，並累加標頭選項內的交換計數欄位。 若封包存在傳輸層標頭則繼續由網路位址埠號轉譯模組執行步驟三，否則內送的封包處理到此結束。 由此可看出，若 IP 標頭選項中交換位址欄位所放置的是私有網域下主機 A 所使擁有的私有 IP 位址 x.x.x.x，則經過網路位址交換模組簡單的交換動作後，封包便可輕易的穿越網路位址轉譯器，無須經過查表與重新計算 IP 標頭檢查碼等費時的動作。
3. 封包進入網路位址埠號轉譯模組，該模組將依封包的來源位址與埠號置換 IP 標頭內的目的位址欄位與傳輸層標頭內的目的埠號欄位。 因此步驟二的位址交換動作將不會影響該模組的運作。 至此便完成所有的內送封包處理程序。

5.4.3 Protocol Description

接下來我們將透過檢視行動網際網路協定的註冊與封包封裝流程來瞭解如何利用上述的機制達成讓行動網際網路協定穿越網路位址轉譯器。 首先在註冊流程的部分會介紹行動端與家代理器如何偵測網路位址轉譯器是否搭載運作中的網路位址交換模組。 而在封包封裝流程的部分將會解釋 IP^{HO} 承載 IP 與 IP 承載 IP 所形成的不對等封裝通道。

■ 註冊訊息流程 (Registration Message Flow)

圖 5-7 所展示的便是帶有型態 252 之 IP 標頭選項的註冊要求與回覆訊息在行動端與家代理器之間的完整流程。 以下便依照封包產生與處理的先後順序詳細解釋每個步驟：

1. 當行動端在外地網域取得一私有網際網路位址 10.113.215.188 作為其配置轉交位址之後，產生一註冊要求訊息(圖中左下角之封包)。 此處與原始註冊要求訊息不一樣的地方在於 IP 標頭之後增加了前述所提及的型態 252 之 IP 標頭選項。 其目的是為了讓家代理器偵測服務行動端的網路位址轉譯器是否有運作的網路位址交換模組。 而該 IP 標頭選項內的交換位址欄位所放置的 IP 位址為 0.0.0.0。
2. 註冊要求訊息在通過網路位址轉譯器時的處理動作就如同上一節所提到的外送封包之處理程序。 因此，家代理器在收到此註冊訊息(圖中右下角之封包)後可用該型態 252 之 IP 標頭選項內的交換計數與交換位址欄位來判斷服務行動端的網路位址轉譯器是否有運作的網路位址交換模組。 而檢查的方法則是交換計數欄位必須大於 0 且交換位址欄位內存放的第一組 IP 位址必須同等於行動網際網路協定註冊要求訊息內的轉交位址，也就是圖例中的 10.113.215.188。

3. 為了讓行動端也知道該網路位址轉譯器具有網路位址轉交模組且家代理器也同意使用 IP^{HO} 承載 IP 的封裝模式，家代理器必須回覆一表示註冊成功的行動網際網路協定註冊回覆訊息並同樣附加型態 252 之 IP 標頭選項(圖中右上角之封包)。 此時標頭選項內的交換位址欄位所放置的 IP 位址則為行動端的轉交位址 10.113.215.188。
4. 同樣的，註冊回覆訊息會依照上一節所提到的內送封包處理程序被網路位址轉譯器所處理。 所以當行動端收到註冊回覆訊息後(圖中左上角之封包)，可視型態 252 之 IP 標頭選項內的交換計數欄位是否大於 0 來決定爾後是否使用 IP^{HO} 承載 IP 的封裝模式。

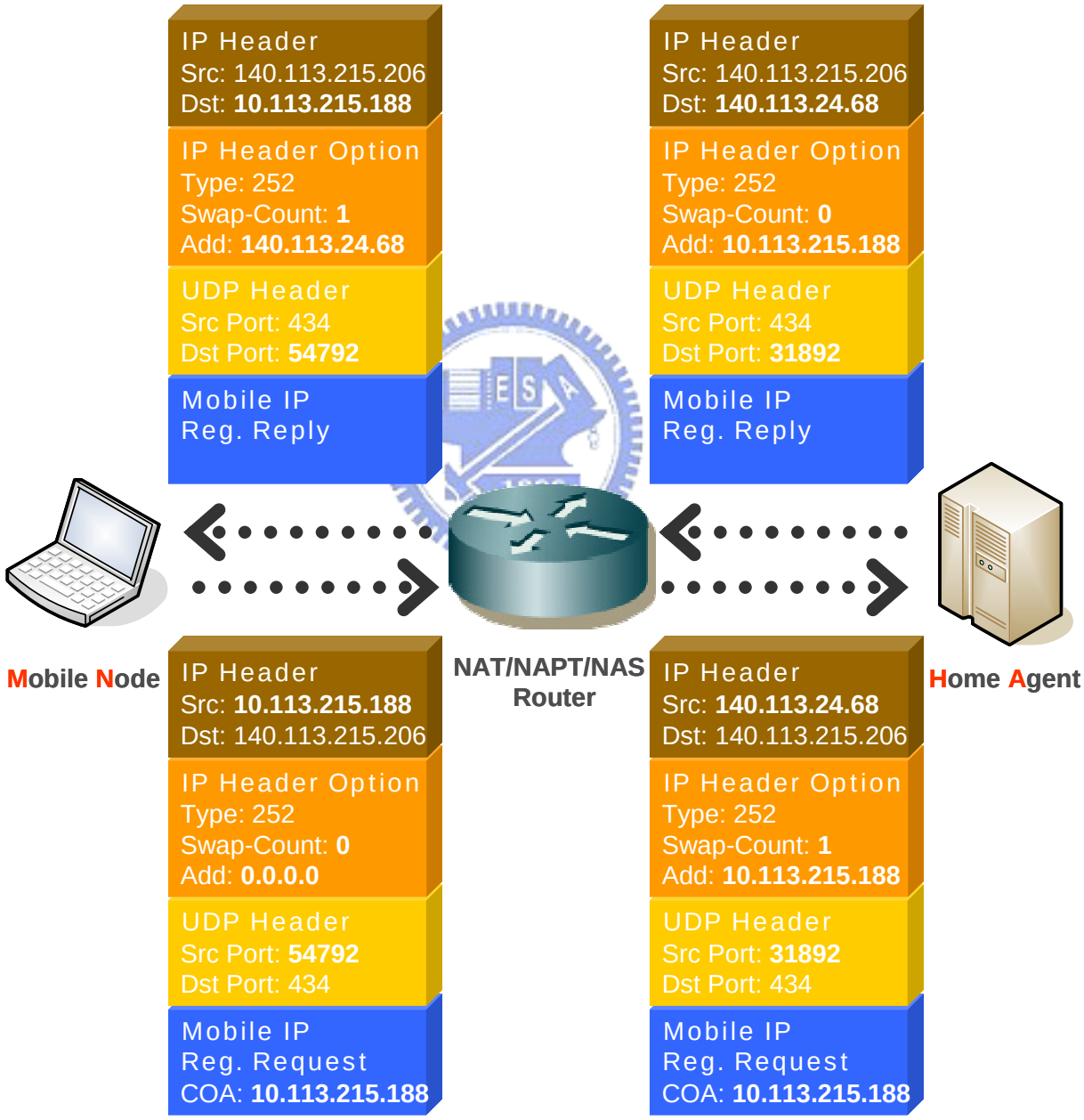

Figure 5 - 7: Registration Request/Reply Message with IP Header Option

■ 數據資料封裝流程 (Datagram Encapsulation Flow)

在行動端與家代理器完成上述的註冊流程後，如果已偵測出網路位址轉譯器具備網路位址交換模組，則家代理器在數據資料的封裝上便會使用 IP^{HO} 承載 IP 的封裝模式。 此時外層 IP 標頭的目的位址為網路位址轉譯器的公有 IP 位址 140.113.24.68，而型態 252 之 IP 標頭選項內的交換位址欄位所放置的 IP 位址則是行動端的私有轉交位址 10.113.215.188。 如此一來，透過網路位址交換模組的位址交換動作後，行動端便可順利的收到經由家代理器所封裝的封包，如圖 5-8 上半部所示。 另一方面，我們先前在 2.1.4 曾提到必須使用反向通道技術來克服搭載防假造過濾器的路由器，因此由行動端所送出的封包也必須經過封裝，雖然在此我們同樣也可以採用 IP^{HO} 承載 IP 的封裝模式，但是這是沒有必要的。 所以在減少封裝上額外訊息標頭負擔的考慮下，對於反向通道依然採用傳統的 IP 承載 IP 的封裝模式，如圖 5-8 下半部所示。 至於拆裝的處理因與原始的拆裝方式相同，便不在此累述。 以上便是前述不對等封裝通道的完整流程解釋。

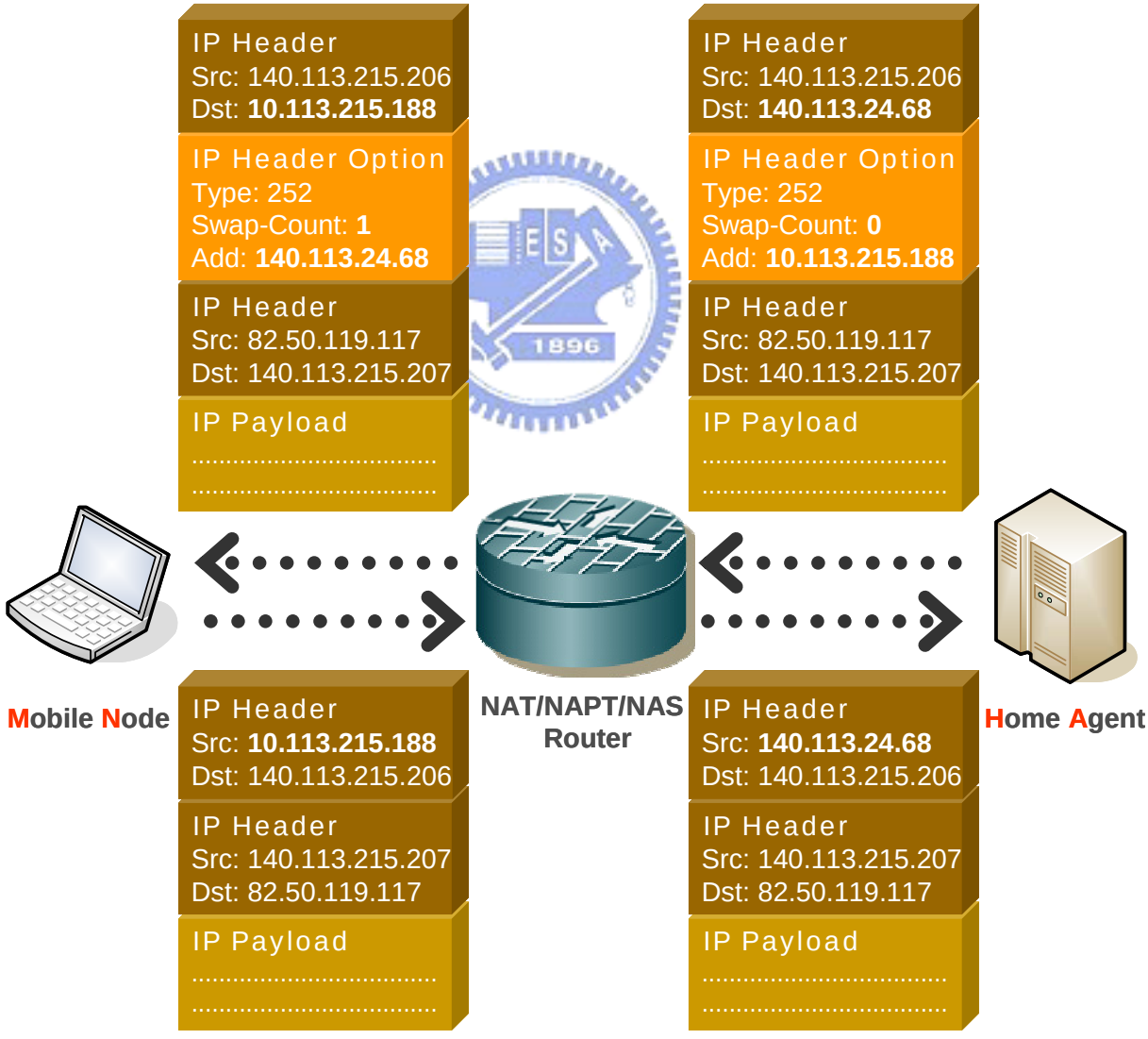

Figure 5 - 8: IP-in-IP^{HO} Tunneling and IP-in-IP Reverse Tunneling

Chapter 6 Implementation Issue

Section 6.1 Implementation of Radio Mobile IP

本節當中將詳細介紹如何實作在第四章所提及的 RIOMIP 軟體套件。 首先在 6.1.1 會說明實作上所使用的軟硬體設備及網路拓撲。 接著在 6.1.2 便是 RIOMIP 內的各個軟體元件實作解說。最後 6.1.3 的部分則為程式的操作實例。

6.1.1 Environment Configuration

在我們的實作環境中，共包含兩個主要的端點，分別是家代理器與行動端。 以下便針對其上所需使用的軟硬體設備及程式做一簡單的介紹後，再說明整體環境的網路設定。

- 硬體需求 (Hardware Requirement)
  - 家代理器 (Home Agent)

    家代理器所使用的機器為一普通的桌上型電腦並配有乙張乙太網路卡，其詳細的硬體規格如下表所列：

Category	Specification
PC	Desktop
Processor	Intel Pentium III (Coppermine) 933 Mhz
Memory	Kingstone PC-133 SDRAM 512 MB
HDD	Seagate ST380021A 7200RPM 80GB
Ethernet	Intel 8255x-based PCI 10/100Mbps

Table 6 - 1: Home Agent Hardware Requirement Table

  - 行動端 (Mobile Node)

    行動端所使用的機器為一筆記型電腦，除了內建的乙太網路卡外，還另外搭配了乙張無線區域網路卡及乙張 GPRS 數據卡，詳細的硬體規格左列於下表：

Category	Specification
PC	Acer TravelMate 354TE Notebook
Processor	Intel Pentium III 850 Mhz
Memory	Transcend PC-100 SDRAM 256 MB
HDD	Hitachi HTS548030M9AT00 5400RPM 30GB
Ethernet	Intel 8255x-based PCI 10/100Mbps
WiFi	3Com OfficeConnect Wireless 11g USB Adapter 11/54Mbps
GPRS	PRETEC CompactGPRS CF 115Kbps

Table 6 - 2: Mobile Node Hardware Requirement Table

- 軟體需求 (Software Requirement)
  - 家代理器 (Home Agent)

    家代理器所使用的軟體如下表所列，其運行的作業系統為 Redhat 版本 9.0 的 Linux，而執行的家代理器程式則為由本實驗室自芬蘭赫爾辛基科技大學所開發的 Dynamics 版本 0.8.1 所修改之支援 RFC 3519 的 Mobile IP 套件。 其中並包含了新增的 UDP 承載 IP 通道裝置。

Category	Specification
Operating System	Redhat Linux 9.0 Kernel 2.4.18-14
Mobile IP Package	HUT Dynamics 0.8.1*
IP-in-IP Tunnel Device	ipip.o
IP-in-UDP Tunnel Device	mipnatdev.o*

Table 6 - 3: Home Agent Software Requirement Table

  - 行動端 (Mobile Node)

    行動端所運行的作業系統為微軟 Windows XP 版本 SP2，而執行的行動端程式則為先前所提及的 RIOMIP 軟體套件。 詳細的軟體規格如下表左列：

Category	Specification
Operating System	Microsoft Windowx XP SP2
Mobile IP Package	RIOMIP 2.3.2.1
Development Tool	Microsoft Visual C++ 6.0 SP6
	Microsoft Platform SDK XP SP2
	Microsoft Device Development Kit 3790
	Compuware DriverStudio 3.0
Testing Software	Microsoft Windows Media Player 10.0
	GlobalSCAPE CuteFTP 7.0 Professional

Table 6 - 4: Mobile Node Software Requirement Table

- 網路拓撲 (Network Topology)

圖 6-1 所展示的便是我們在以下實作過程中所使用的網路環境設定。 其中家網域及外地網域的有線/無線區域網路，也就是網路前置碼為 140.113.0.0 之網路為交通大學所屬。 而通訊端所在的網域 140.109.0.0，則是屬於中央研究院所有。 至於 GPRS 服務的系統業者則為台灣大哥大。

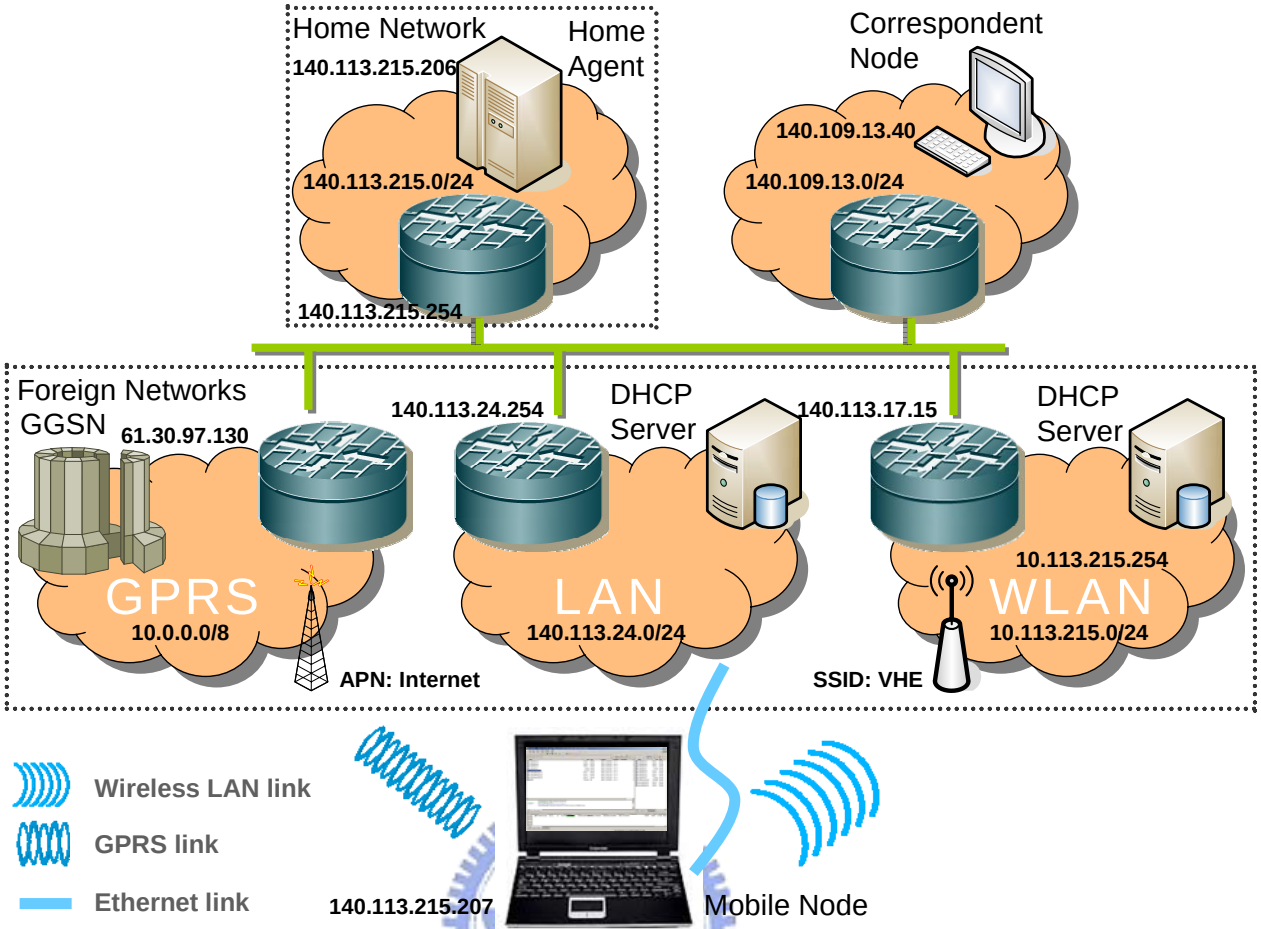

Figure 6 - 1: Radio Mobile IP Implementation Network Topology

6.1.2 Software Development

以下便針對 RIOMIP 內各個軟體元件在實作上的重點部分做一介紹：

- NdisFlt Driver

該 NDIS 攔截驅動程式原自於 Vlad G.所開發的"Simple NDIS Hooking Based Firewall for NT4/2000" [34]。 它是一個簡單的封包過濾防火牆，包含了通用的 NDIS 攔截驅動程式(ndis_hk)與封包過濾驅動程式(ndis_flt)。 NDIS 攔截驅動程式允許取得介面卡的列表並安裝核心模式的回呼函式來允許或拒絕封包自 TCP/IP 協定堆疊的進出，而封包過濾驅動程式則是用來解析存放過濾規則的檔案並包含用來做封包過濾的回呼函式。 由於我們需要的功能是取得完整的封包內容並加以過濾，因此我們便捨棄封包過濾驅動程式的部分並對原本的 NDIS 攔截驅動程式加以修改以符合我們的需求。 至於修改的部分主要可分為過濾封包函式及回呼函式介面兩個方面，以下便針對這兩個部分逐一解說：

- 過濾封包函式 (Filter Packet Function)

誠如先前章節所提，當 NDIS 攔截驅動程式載入系統後會置換掉原本 NDIS 介面用來傳送與接收封包的函式指標，其用意就是在這些取代函式內呼叫此過濾封包函式來執行

封包的擷取與過濾動作。 因此，本函式負責的部分便是將完整的封包內容放入預先配置的記憶體空間後傳入已註冊的回呼函式。 而根據 DDK [31] 文件的說明，一個封包可能散落在實體記憶體中不連續的位置，如下圖所示，所以在呼叫回呼函式之前，我們必須透過 NDIS 提供的函式將封包還原成一連續的記憶體空間才能供其它註冊回呼函式的核心模式驅動程式來使用。

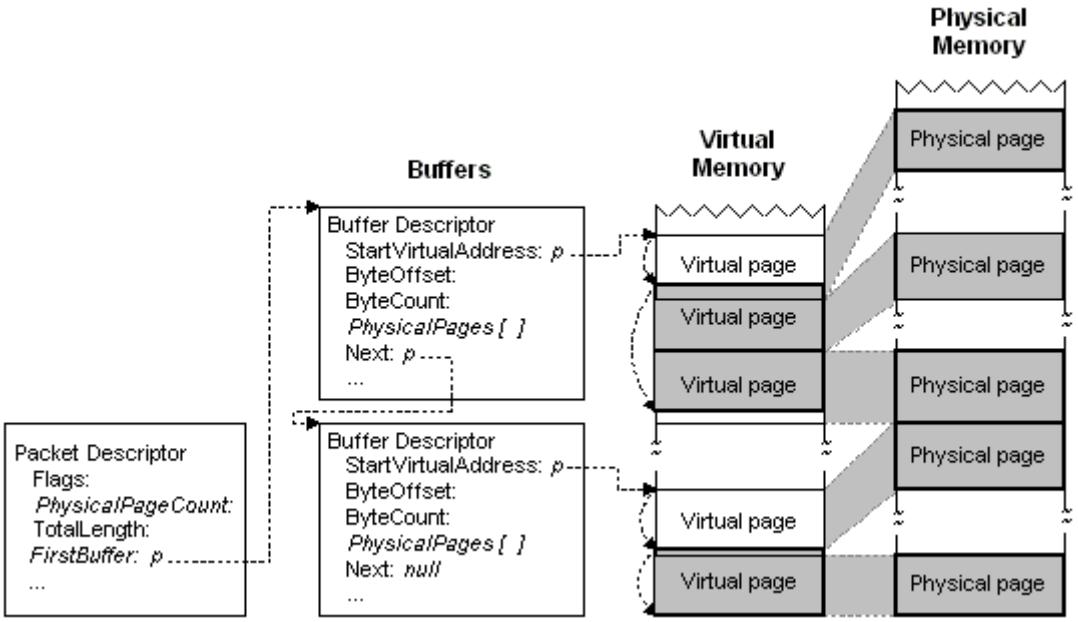

Figure 6 - 2: NDIS Packet Structure

圖 6-3 便是此函式的詳細內容。 傳入函式的參數包括指示封包方向(進或出)的變數 direction，封包經由的介面編號變數 iface 以及封包的指標 ndis_packet。 函式一開始的 if 判斷句表示我們只處理 IP 或是 ARP 的封包。 接下來在檢查封包正確性的 if 判斷句之後的 while 函式便是用來將完整的封包存入預先配置好的記憶體空間變數 packet 內。 最後，如果發現有已註冊的回呼函式，便將包含上述 packet 等變數作為參數傳入該函式。至於該回呼函式的回傳布林值則代表是否允許該封包的通過，TRUE 代表放行該封包，FALSE 則表示將該封包丟棄，而過濾封包函式本身的預設值為 TRUE。

```
BOOLEAN filter_packet(int direction, int iface, PNDIS_PACKET ndis_packet) {
  BOOLEAN result = TRUE;
  PNDIS_BUFFER buffer;
  UINT packet_len, packet_offset, buffer_len, buffer_offset, hdr_len;
  PVOID pointer;

  NdisQueryPacket(ndis_packet, NULL, NULL, &buffer, &packet_len);
  if ((buffer == NULL) || (packet_len < sizeof(struct ether_hdr))) { return TRUE; }
  NdisQueryBufferSafe(buffer, &pointer, &buffer_len, NormalPagePriority);
  if ((pointer == NULL) || (buffer_len < sizeof(struct ether_hdr))) { return TRUE; }
  if (ntohs(((struct ether_hdr *)pointer)->ether_type) == ETHERTYPE_IP ||
     ntohs(((struct ether_hdr *)pointer)->ether_type) == ETHERTYPE_ARP) {
    buffer_offset = 0;
    packet_offset = 0;
    if ((ntohs(((struct ether_hdr *)pointer)->ether_type) == ETHERTYPE_IP &&
       (packet_len < sizeof(struct ether_hdr)+sizeof(struct ip_hdr) || packet_len > 1514)) ||
       (ntohs(((struct ether_hdr *)pointer)->ether_type) == ETHERTYPE_ARP &&
       (packet_len < sizeof(struct ether_hdr)+sizeof(struct arp_pkt) || packet_len > 1514))){
       return TRUE;
    }
    while (packet_offset < packet_len) {
      while (buffer_offset < buffer_len) {
        packet[packet_offset++] = *((PUCHAR)pointer+buffer_offset++);
      }
      if (packet_offset < packet_len) {
        NdisGetNextBuffer(buffer, &buffer);
        if (buffer == NULL) break;
        NdisQueryBufferSafe(buffer, &pointer, &buffer_len, NormalPagePriority);
        if (pointer == NULL || buffer_len <= 0) break;
        buffer_offset = 0;
      }
    }
    if (packet_offset == packet_len) {
      if (g_has_callbacks) {
        KIRQL irql;
        KeAcquireSpinLock(&guard, &irql);
        if (g_callbacks.hook_packet != NULL)
          result = g_callbacks.hook_packet(direction, iface, packet, packet_offset);
        KeReleaseSpinLock(&guard, irql);
      } else {
        result = TRUE;
      }
    }
  }
  return result;
}
```

Figure 6 - 3: NdisFlt Driver Filter Packet Function

- 回呼函式介面 (Callback Function Interface)

至於回呼函式的介面就如圖 6-4 所示，它是一個函式指標的形式。 凡欲向 NdisFlt 驅動程式註冊回呼函式的其它核心模式驅動程式(如稍後會提到的 IpFlt 驅動程式)都必須遵守這樣的函式定義。 參數部分依次為指示封包方向的變數 direction，封包經由的介面編號變數 iface，封包所在位址的指標 packet 以及表明封包長度的變數 size。 而在回傳值部分則必須是 TRUE 或 FALSE 的布林值。

```
struct filter_callbacks
{
   BOOLEAN  (*hook_packet)(int direction, int iface, void *packet, UINT size);
};
```

Figure 6 - 4: NdisFlt Driver Callback Function Interface

■ IpFlt Driver

IpFlt 驅動程式的製作修改自 Hollis Technology Solutions 所提供的"IpHook" [35]。 其實作上可分為兩個部分，首先在核心模式的驅動程式方面所運用的便是 2.3.3 所提及的 Windows 2000 過濾攔截驅動程式，而在用戶模式下的應用程式方面則利用該驅動程式來擷取 TCP/IP 堆疊中所有進出的封包並提供圖形化的介面讓使用者觀察。 然而在先前我們曾提過使用 Windows 2000 過濾攔截驅動程式的缺點在於取得的封包內容僅限於各個協定的標頭部分而非包含完整的承載數據資料，於是我們便著手修改 IpHook 的核心驅動程式讓它改用上述的 NDIS 攔截驅動程式而非 Windows 2000 過濾攔截驅動程式來彌補這項缺失。 此外，由於在實作上並不需要 IpHook 用戶模式部分的程式，因此也將該部分加以捨棄。 至於詳細的修改內容可分為以下幾個部分：

- 裝置控制 (Device I/O Control)

  我們一共設計了九個 DeviceIOControl 命令來控制此驅動程式的運作，其名稱與用途如下表左列：

IOCTL Code	Description
IPFLT_ENABLE	啟始 IpFlt 驅動程式的運作
IPFLT_DISABLE	停止 IpFlt 驅動程式的運作
IPFLT_PUSH_BUFFER	傳遞用來存放封包的暫存空間給 IpFlt 驅動程式
IPFLT_CANCEL_BUFFER	取消 IpFlt 驅動程式內用來存放封包的暫存空間
IPFLT_ENUM_ADAPTERS	取得已向 NDIS 完成註冊的介面卡資訊
IPFLT_SET_IPADDR	設定 IpFlt 驅動程式過濾的 IP 位址
IPFLT_SET_IPIF	設定 IpFlt 驅動程式過濾的介面卡
IPFLT_FLT_ARP	設定 IpFlt 驅動程式是否過濾 ARP 封包
IPFLT_FLT_IP	設定 IpFlt 驅動程式是否過濾 IP 封包

Table 6 - 5: IpFlt Driver Device I/O Control Table

- 回呼函式 (Callback Function)

  圖 6-5 便是 IpFlt 驅動程式內用來向 NdisFlt 驅動程式註冊的回呼函式，參數及回傳值的部分皆遵守上述回呼函式介面的規範。 同樣的，該函式只針對 ARP 以及版本 4 的 IP 封包做處理，且依照封包進出的方向各有不同的動作。 在傳送的 IP 封包方面，我們視 IPFLT_FLT_IP 命令的設定只攔截以 IPFLT_SET_IPIF 命令所設定的介面卡上來源位址為用 IPFLT_SET_IPADDR 命令所指定的 IP 位址之數據資料。 但有兩個例外，一是行動網際網路協定之封包，也就是 UDP 目的埠號為 434 之封包，二則為網際網路控制訊息協定路由器請求訊息。 而在發送的 ARP 封包方面，所有以 IPFLT_SET_IPADDR 命令設定之 IP 位址所發送的 ARP 請求與回覆訊息都會視 IPFLT_FLT_ARP 命令的設定被丟棄或允許。 另一方面，在接收 IP 封包的處理上，同樣視 IPFLT_FLT_IP 命令的設定僅攔截行動

網際網路協定的封包以及網際網路控制訊息協定路由器公告訊息。 此外，視 IPFLT_FLT_ARP 命令的設定將會丟棄詢問 IPFLT_SET_IPADDR 命令所指定的 IP 位址之 ARP 請求訊息。 最後，所有由 IpFlt 驅動程式所攔截的 IP 封包將被放置在稍後會提到的 iphookdata 資料結構內給提供此封包暫存空間的用戶模式應用程式做後續的處理動作。

```
BOOLEAN IpHooker(int Direction, int InterfaceIndex, void *Packet, unsigned int PacketLength)
{
 ULONG sequence = ipGlobal.getSequence();
 ETHHeader *eth = (ETHHeader *)Packet;
 IPHeader *ip = (IPHeader *)((PUCHAR)Packet+sizeof(ETHHeader));
 ARPHeader *arp = (ARPHeader *)((PUCHAR)Packet+sizeof(ETHHeader));

 if ( ntohs(eth->eth_type) == 0x0806 ||
     (ntohs(eth->eth_type) == 0x0800 && (ip->ip_verlen & 0xf0) == 0x40)) {
  if (Direction == DIRECTION_OUT) {
    if (ntohs(eth->eth_type) == 0x0800)       {
      if ( !ipGlobal.GetFltIPflag() ) { return TRUE; }
      if ( ipGlobal.GetFltIPif() != 0 && ipGlobal.GetFltIPif() != InterfaceIndex ) { return TRUE; }
      if ( (ip->ip_src & ipGlobal.GetFltIPaddr()) != ipGlobal.GetFltIPaddr() ) { return TRUE; }
      if ( ip->ip_protocol == 17 &&
           (PacketLength > sizeof(ETHHeader) + (ip->ip_verlen & 0x0f)*4 + sizeof(UDPHeader)) &&
           ((UDPHeader *)((PUCHAR)ip+(ip->ip_verlen & 0x0f)*4))->udp_dport == 0xb201 ) { return TRUE; }
      if ( ip->ip_protocol == 1 &&
           (PacketLength > sizeof(ETHHeader) + (ip->ip_verlen & 0x0f)*4 + sizeof(ICMPHeader)) &&
           ((ICMPHeader *)((PUCHAR)ip+(ip->ip_verlen & 0x0f)*4))->icmp_type == 10 &&
           ((ICMPHeader *)((PUCHAR)ip+(ip->ip_verlen & 0x0f)*4))->icmp_code == 0) { return TRUE; }
    } else {
      if (ntohs(arp->arp_htype) == 0x0001 && ntohs(arp->arp_ptype) == 0x0800 &&
          arp->arp_hsize == 6 && arp->arp_psize == 4 &&
          *(PULONG)&arp->arp_sip[0] == ipGlobal.GetFltIPaddr() &&
         (ntohs(arp->arp_op) == 0x0001 || ntohs(arp->arp_op) == 0x0002)) {
        if (ipGlobal.GetFltARPflag()) return { FALSE; }
      }
    }
  } else { // DIRECTION_IN
    if (ntohs(eth->eth_type) == 0x0800)       {
      if ( ip->ip_src == ipGlobal.GetFltIPaddr()) { return FALSE; }
      if ( ip->ip_protocol != 1 && ip->ip_protocol != 4 && ip->ip_protocol != 17) { return TRUE; }
      if ( ip->ip_protocol == 1 &&
           PacketLength > sizeof(ETHHeader) + (ip->ip_verlen & 0x0f)*4 + sizeof(ICMPHeader) &&
           ((ICMPHeader *)((PUCHAR)ip+(ip->ip_verlen & 0x0f)*4))->icmp_type != 9) { return TRUE; }
      if ( ip->ip_protocol == 17 &&
           PacketLength > sizeof(ETHHeader) + (ip->ip_verlen & 0x0f)*4 + sizeof(UDPHeader) &&
           ((UDPHeader *)((PUCHAR)ip+(ip->ip_verlen & 0x0f)*4))->udp_dport != 0xb201) { return TRUE; }
    } else {
      if ( ntohs(arp->arp_htype) == 0x0001 &&  ntohs(arp->arp_ptype) == 0x0800 &&
           arp->arp_hsize == 6 && arp->arp_psize == 4 && ntohs(arp->arp_op) == 0x0001) {
        if (*(PULONG)arp->arp_sip == ipGlobal.GetFltIPaddr()) { return FALSE; }
        if (*(PULONG)arp->arp_tip == ipGlobal.GetFltIPaddr()) {
          if (ipGlobal.GetFltARPflag()) { return FALSE; }
        }
      }
    }
  }

  if (htons(eth->eth_type) == 0x0800) {
    PIPHOOK_DATA iphookdata = ipGlobal.getBuffer();
    if (iphookdata) {
      ULARGE_INTEGER ticks;
      KeQueryTickCount((PLARGE_INTEGER)&ticks);
      iphookdata->timestamp = ticks.QuadPart;
      iphookdata->tag = IPHOOK_DATA_TAG;
      iphookdata->sequence = sequence;
      if (PacketLength > 0) {
        unsigned int size = 0;
        do {
          iphookdata->packet.eth_payload[size] = *((PUCHAR)Packet + size++);
        } while(size < PacketLength);
      }
      iphookdata->dataLength = PacketLength;
      iphookdata->direction = Direction;
      iphookdata->ifIndex = InterfaceIndex;
    }
    return FALSE;
  }
 }
 return TRUE;
}
```

Figure 6 - 5: IpFlt Driver Callback Function

- 封包暫存 (Packet Buffer)

由 IpFlt 驅動程式所攔截的封包皆放置在 IPHOOK_DATA 的資料結構內，用戶模式下的應用程式透過表 6-5 中的 IPFLT_PUSH_BUFFER 命令將預留好一連串 IPHOOK_DATA 所需要的空間之 IPHOOK_BUFFER 資料結構傳入驅動程式內。 驅動程式便以 4.4.4 所提及的逆向呼叫方式將這些封包的暫存空間以鏈結串列(Linked List)的形式保存下來。 資料結構的詳細內容如圖 6-6 所示：

```
typedef struct {
  ULONG           tag;
  ULONG           sequence;
  ULONGLONG       timestamp;
  ULONG           direction;
  int             ifIndex;
  ULONG           dataLength;
  ETHFrame        packet;
} IPHOOK_DATA, *PIPHOOK_DATA;

typedef struct {
  ULONG           tag;
  ULONG           entries;
  ULONG           valid;
  IPHOOK_DATA     buffer[1];
} IPHOOK_BUFFER, *PIPHOOK_BUFFER;
```

Figure 6 - 6: IpFlt Driver Packet Buffer Data Structure

當 IpFlt 驅動程式呼叫圖 6-5 中的 getBuffer 函式時，便會自鏈結串列取得尚有空間的 IPHOOK_BUFFER 資料結構內的一個 IPHOOK_DATA。 並將其內的各個欄位填入適當的值之後更新該 IPHOOK_BUFFER 資料結構內用來記錄有效 IPHOOK_DATA 數量的欄位 valid。 如發現 valid 欄位的數值已到達 entries 欄位的數值，則表示該 IPHOOK_BUFFER 無法存放更多的 IPHOOK_DATA，此時 IpFlt 驅動程式將以 IoComplete 命令指示用戶模式的應用程式取回該資料。 然而，如果一味的等待封包填滿至最大量可能造成封包處理上的延遲。 因此，當 IpFlt 驅動程式在啟始時會額外開出一支核心模式的執行緒(Thread)負責每五百萬分之一秒(5 ms)以 IoComplete 命令結束已存放資料的 IPHOOK_BUFFER 之使用，如此一來用戶模式的應用程式便可即時的得到由 IpFlt 驅動程式所攔截的封包。

- Mobile IP Service Intermediate Driver

Mobile IP 服務中間層驅動程式是透過 Compuware 公司的 Driver Studio 所設計的標準中間層驅動程式，該驅動程式原本只是純粹地傳遞協定層驅動程式與迷你連接埠驅動程式間的訊息。 因此，為了解決媒體感應所造成的問題，我們修改了狀態指示與詢問所對應的處理函式，如圖 6-7 所示。

- 狀態指示處理函式 (OnStatusIndication)

  此函式內主要的動作是要阻止底層的迷你連接埠驅動程式傳送媒體連結中斷的訊息 NDIS_STATUS_MEDIA_DISCONNECT 給上層的協定層驅動程式。 因此，當中間層驅動程式接收到該訊息後，會以 NDIS_STATUS_MEDIA_CONNECT 來取代之，而真正的媒體連結狀態會存在內部所使用的變數 m_NdisMediaState 內。

- 詢問處理函式 (OnQuery)

  而在處理協定層驅動程式利用 OID_GEN_MEDIA_CONNECT_STATUS 來詢問迷你連接埠驅動程式關於媒體的連結狀態上，中間層驅動程式將無視底層媒體真正的連結狀態為何，一律以連結狀態正常的 NdisMediaStateConnected 訊息回覆詢問的協定層驅動程式。 只有在收到 OID_VEN_MEDIA_CONNECT_STATUS 的詢問時，中間層驅動程式才會回覆存在變數 m_NdisMediaState 內真正的媒體連結狀態。

```
NDIS_STATUS MIPAdapter::OnStatusIndication (NDIS_STATUS Status, IN OUT PVOID* StatusBuffer, UINT* StatusBufferSize)
{
  TRACE("MIPAdapter::OnStatusIndication() 0x%X\n", Status);
  UNREFERENCED_PARAMETER(StatusBuffer);
  UNREFERENCED_PARAMETER(StatusBufferSize);
  if (Status == NDIS_STATUS_MEDIA_CONNECT)
  {
    m_NdisMediaState = NdisMediaStateConnected;
    return Status;
  }
  if (Status == NDIS_STATUS_MEDIA_DISCONNECT)
  {
    m_NdisMediaState = NdisMediaStateDisconnected;
    return NDIS_STATUS_MEDIA_CONNECT;
  }
  return Status;
}

NDIS_STATUS MIPAdapter::OnQuery (KNdisRequest& Request, NDIS_STATUS ReturnedStatus)
{
  TRACE("MIPAdapter::OnQuery() %s\n", Request.Text());
  if (OID_TCP_TASK_OFFLOAD == Request.DATA.QUERY_INFORMATION.Oid)
    return NDIS_STATUS_NOT_SUPPORTED;
  if (OID_GEN_MEDIA_CONNECT_STATUS == Request.DATA.QUERY_INFORMATION.Oid &&
      ReturnedStatus == NDIS_STATUS_SUCCESS)
  {
    *(PNDIS_MEDIA_STATE)Request.DATA.QUERY_INFORMATION.InformationBuffer = NdisMediaStateConnected;
  }
  if (OID_VEN_MEDIA_CONNECT_STATUS == Request.DATA.QUERY_INFORMATION.Oid)
  {
    if (Request.DATA.QUERY_INFORMATION.InformationBufferLength < 1)
    {
      Request.DATA.QUERY_INFORMATION.BytesNeeded = 1;
      ReturnedStatus = NDIS_STATUS_INVALID_LENGTH;
    }
    else
    {
      *(PNDIS_MEDIA_STATE)Request.DATA.QUERY_INFORMATION.InformationBuffer = m_NdisMediaState;
      Request.DATA.QUERY_INFORMATION.BytesWritten = 1;
      ReturnedStatus = NDIS_STATUS_SUCCESS;
    }
  }
  return ReturnedStatus;
}
```

Figure 6 - 7: Mobile IP Service Intermediate Driver Internal Functions

透過圖 6-8 與圖 6-9 的比較可以發現，在套用 Mobile IP 服務中間層驅動程式之後，如將網路線拔離網路卡，雖然以 ipconfig 命令還是可以見到 Media disconnected 的訊息，但是用 route print 的命令來檢視系統的路由表可發現，網路卡上所設定的 IP 位址與閘道器等資訊仍皆保留住，

不像在未套用此驅動程式前會發生路由表被清空的情形。 也就是說，在此情況下我們成功地讓 TCP/IP 協定堆疊認為底層的網路卡仍可運作。

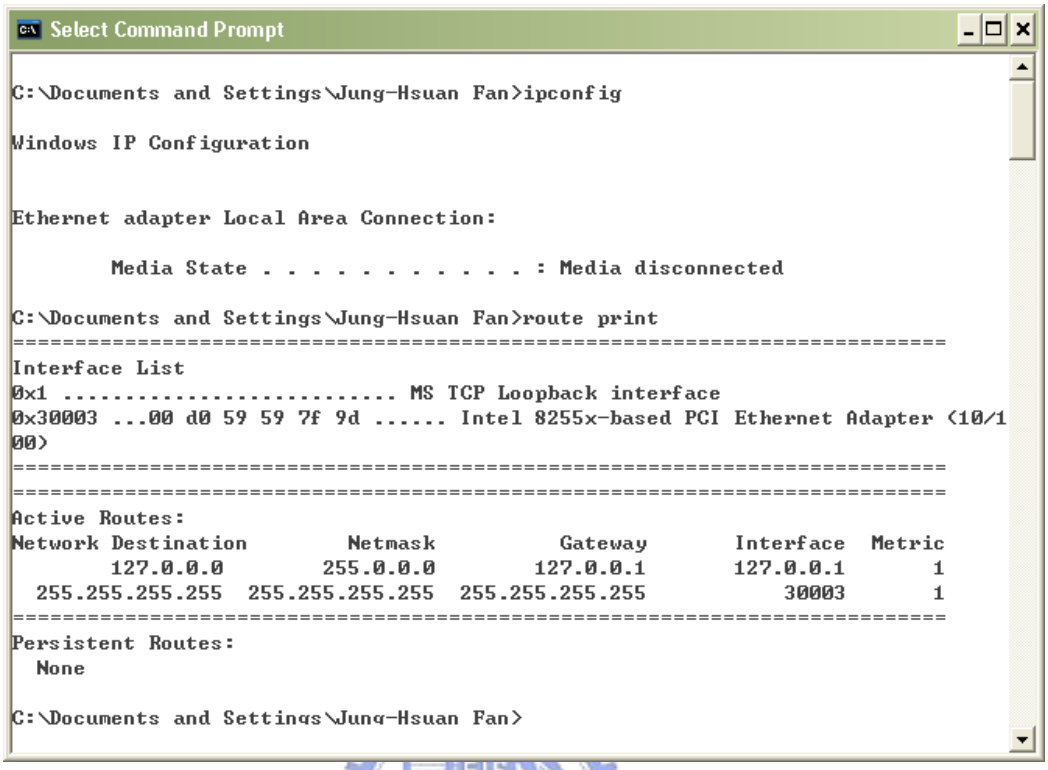

Figure 6 - 8: Routing Table before applying Mobile IP Service Intermediate Driver

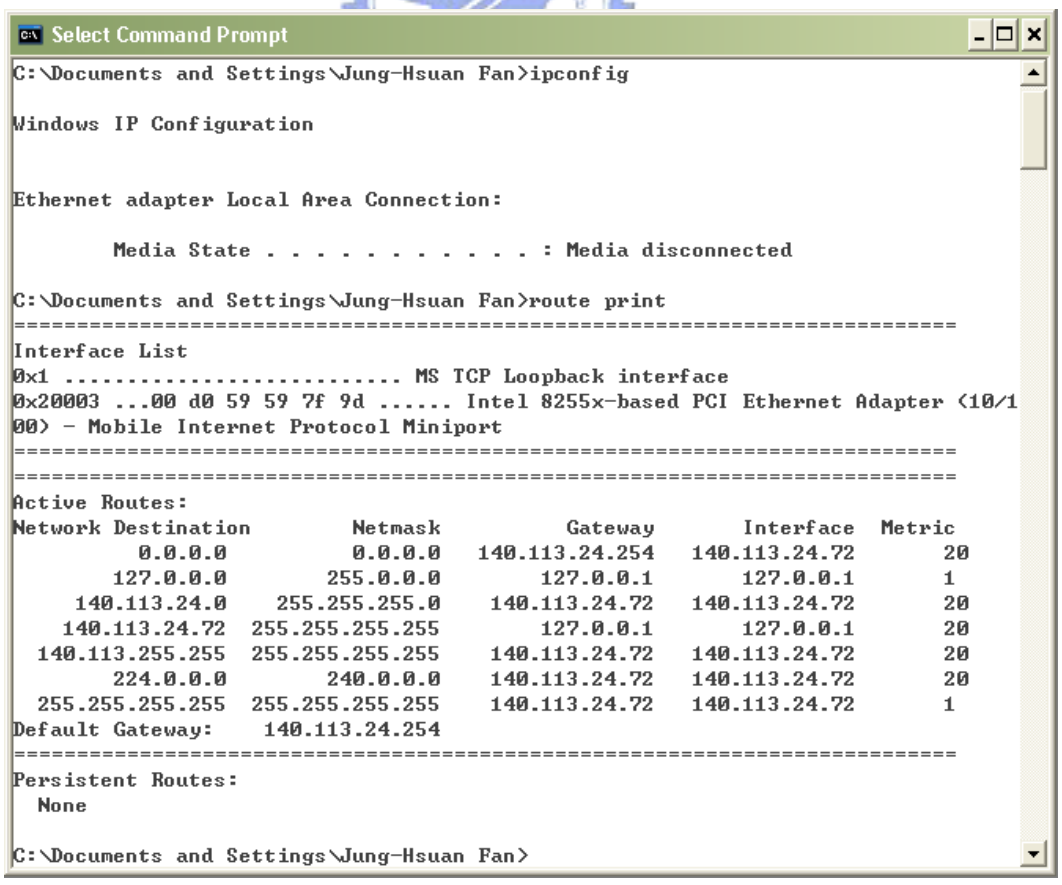

Figure 6 - 9: Routing Table after applying Mobile IP Service Intermediate Driver

NdisProt Protocol Driver

在微軟的驅動程式開發套件(Driver Development Kit, DDK)中所提供的範例程式 ndisprot 便是一個標準的 NDIS 協定驅動程式。 它支援在用戶模式下使用 ReadFile/WriteFile 函式來傳送與接收乙太網路的訊框，並且還可用 DeviceIOControl 命令來取得與設定底層迷你連接埠驅動程式內的物件識別元。 換句話說，在功能上該範例程式已經符合我們大部分的需求，除了缺乏接收網路媒體事件通知外。 因此，我們所要做的便是補足這項功能。 同樣地，我們分成兩個部分來討論該部分的實作：

- 裝置控制 (Device I/O Control)

  我們在原有的五個 DeviceIOControl 命令之外新增了兩個用來控制接收網路媒體狀態改變的事件通知訊息(下表中以星號註記者)，所有的控制命令名稱與用途如表 6-6 左列：

IOCTL Code	Description
IOCTL_NDISPROT_OPEN_DEVICE	指定欲動作的網路配接卡
IOCTL_NDISPROT_QUERY_OID_VALUE	查詢物件識別元
IOCTL_NDISPROT_SET_OID_VALUE	設定物件識別元
IOCTL_NDISPROT_QUERY_BINDING	查詢 NdisProt 驅動程式可動作的網路配接卡
IOCTL_NDISPROT_BIND_WAIT	等待 NdisProt 驅動程式完成啟始動作
IOCTL_NDISPROT_PUT_TRIGGER*	傳送存放媒體事件的暫存空間
IOCTL_NDISPROT_REMOVE_TRIGGER*	取消存放媒體事件的暫存空間

Table 6 - 6: NdisProt Protocol Driver Device I/O Control Table

用戶模式的應用程式在使用此 NdisProt 協定驅動程式時的第一個動作便是使用 IOCTL_NDISPROT_BIND_WAIT 命令等待驅動程式完成啟始的動作後，才可以 IOCTL_NDISPROT_QUERY_BINDING 命令查詢可動作的網路配接卡之名稱，並將欲動作網路配接卡的名稱以 IOCTL_NDISPROT_OPEN_DEVICE 命令向 NdisProt 協定驅動程式指定後，才可執行其他的控制命令。 IOCTL_NDISPROT_PUT_TRIGGER 命令的動作就如同上述 IpFlt 驅動程式內的 IPFLT_PUSH_BUFFER，以下便詳加解釋。

- 指示觸發 (Indication Trigger)

  用戶模式的應用程式為了即時取得底層迷你連接埠驅動程式送給協定驅動程式的網路媒體狀態改變事件通知，它會預先配置好一些用來存放該訊息的記憶體空間透過上述的裝置控制命令 IOCTL_NDISPROT_PUT_TRIGGER 傳入 NdisProt 協定驅動程式。 當 NdisProt 協定驅動程式接收到該命令後，便由其內的 NdisProtTrigger 函式負責處理，如圖 6-10 所示。 該函式首先會做一些正確性的檢查，沒問題之後便將該記憶體空間加到名為 PendedTrigger 的鏈結串列內供往後的使用。 最後以 IoMarkIrpPending 函式告知用戶

模式的應用程式該控制命令正在處理中即完成處理的動作。 因此，用戶模式的應用程式必須開出額外的執行緒來等待該控制命令的完成，也就是當有媒體狀態改變的發生。

```
NTSTATUS NdisProtTrigger(
  IN PIRP                             pIrp,
  IN PIO_STACK_LOCATION               pIrpSp,
  IN PNDISPROT_OPEN_CONTEXT           pOpenContext )
{
  NTSTATUS            NtStatus;
  do
  {
    NPROT_STRUCT_ASSERT(pOpenContext, oc);
    if (pIrpSp->Parameters.DeviceIoControl.OutputBufferLength < sizeof(NDIS_STATUS))
    {
      NDIS_STATUS_TO_NT_STATUS(NDIS_STATUS_BUFFER_TOO_SHORT,&NtStatus);
      break;
    }
    NPROT_ACQUIRE_LOCK(&pOpenContext->Lock);
    if (!NPROT_TEST_FLAGS(pOpenContext->Flags, NUIOO_BIND_FLAGS, NUIOO_BIND_ACTIVE))
    {
      NPROT_RELEASE_LOCK(&pOpenContext->Lock);
      NDIS_STATUS_TO_NT_STATUS(NDIS_STATUS_FAILURE,&NtStatus);
      break;
    }

    NPROT_INSERT_TAIL_LIST(&pOpenContext->PendedTriggers, &pIrp->Tail.Overlay.ListEntry);
    NPROT_REF_OPEN(pOpenContext);  // pended trigger
    pOpenContext->PendedTriggerCount++;

    pIrp->Tail.Overlay.DriverContext[0] = (PVOID)pOpenContext;
    IoMarkIrpPending(pIrp);
    IoSetCancelRoutine(pIrp, NdisProtCancelTrigger);

    NPROT_RELEASE_LOCK(&pOpenContext->Lock);
    NtStatus = STATUS_PENDING;
  }
  while (FALSE);

  if (NtStatus != STATUS_PENDING)
  {
    NPROT_ASSERT(NtStatus != STATUS_SUCCESS);
    pIrp->IoStatus.Information = 0;
    pIrp->IoStatus.Status = NtStatus;
    IoCompleteRequest(pIrp, IO_NO_INCREMENT);
  }
  return (NtStatus);
}
```

Figure 6 - 10: NdisProt Protocol Driver I/O Control Handle Function

承上所述，當底層迷你連接埠驅動程式以 NdisMIndicateStatus 函式通知上層的協定驅動程式發生媒體狀態改變的時候，NdisProt 協定驅動程式便呼叫圖 6-11 的 ndisprotServiceTrigger 來負責將發生的事件通知用戶模式的應用程式。 其作法為在 PendedTrigger 鏈結串列內找尋方才存入的任一可用記憶體空間，並將所發生的事件指示 GeneralStatus 變數之值存入其中。 最後以 IoCompleteRequest 完成該尚在處理中的控制命令。 如此一來，用戶模式的應用程式便可從已完成的控制命令中取得網路媒體狀態改變的事件通知。

```
VOID ndisprotServiceTriggers(
  IN PNDISPROT_OPEN_CONTEXT      pOpenContext,
  IN NDIS_STATUS                 GeneralStatus)
{
  PIRP               pIrp;
  PLIST_ENTRY        pIrpEntry;
  BOOLEAN            FoundPendingIrp;

  NPROT_REF_OPEN(pOpenContext);
  do
  {
    if (NPROT_IS_LIST_EMPTY(&pOpenContext->PendedTriggers))
    {
      break;
    }
    FoundPendingIrp = FALSE;
    pIrpEntry = pOpenContext->PendedTriggers.Flink;
    while (pIrpEntry != &pOpenContext->PendedTriggers)
    {
      pIrp = CONTAINING_RECORD(pIrpEntry, IRP, Tail.Overlay.ListEntry);
      if (IoSetCancelRoutine(pIrp, NULL))
      {
        NPROT_REMOVE_ENTRY_LIST(pIrpEntry);
        FoundPendingIrp = TRUE;
        break;
      } else {
        pIrpEntry = pIrpEntry->Flink;
      }
    }
    if (FoundPendingIrp == FALSE)
    {
      break;
    }

    *(PNDIS_STATUS)pIrp->AssociatedIrp.SystemBuffer = GeneralStatus;

    pIrp->IoStatus.Status = STATUS_SUCCESS;
    pIrp->IoStatus.Information = sizeof(NDIS_STATUS);
    IoCompleteRequest(pIrp, IO_NO_INCREMENT);
    NPROT_DEREF_OPEN(pOpenContext);
    pOpenContext->PendedTriggerCount--;
  }
  while (FALSE);
  NPROT_DEREF_OPEN(pOpenContext);    // temp ref - service reads
}
```

Figure 6 - 11: NdisProt Protocol Driver NdisMIndicateStatus Handle Function

- RIOMIP Mobility Management Client

RIOMIP 行動管理客戶端程式是用微軟基礎類別(Microsoft Foundation Class, MFC)函式庫所開發的用戶模式視窗應用程式。 主要為實現了行動網際網路協定堆疊的運作，以下便針對實作上的細節部分做解說。 不過在此之前先說明接下來圖例中的環境設定，如圖 6-1 的網路拓樸，我們將以有線區域網路與無線區域網路間的漫遊作為解說的範例。

- IP 位址之設定 (IP Address Configuration)

承接 4.4.5 的介紹，在我們的設計上將可供運作的網路卡區分為主副配接卡。 一般來說，主配接卡可設定成有線或無線的區域網路卡，而副配接卡則可是主配接卡之外的

任一網路卡，其中包含各式的撥號數據卡。 由於我們所運行的行動網際網路協定為配置位址轉交模式，因此不管是主或副配接卡在程式啟始時都會被設定成自動取得 IP 位址，也就是透過 DHCP 伺服器來取得合適的 IP 位址。 然而，根據行動網際網路協定的規範，行動端上的應用程式所送出的封包必須皆以家位址作為來源 IP 位址。 換句話說，在不修改應用程式的前提下，行動端的家位址必須存在於系統內其中的一張網路卡上才能供應用程式來使用。 因此，針對這樣的問題我們用了 IP 別名(Alias)的技巧來加以克服，也就是我們讓主配接卡擁有兩個 IP 位址。 透過圖 6-12 我們可以更清楚地瞭解實際的狀況，首先圖中的主配接卡為有線區域網路卡而副配接卡則為無線區域網路卡，兩者的自動取得 IP 功能皆被打開。 其次，在主配接卡的部分可以發現到它擁有兩個 IP 位址，一是家位址 140.113.215.207，二則是自動取得的 IP 位址 169.254.91.126。 而副配接卡的 IP 位址只有一個，也就是透過 DHCP 協定所自動取得的 10.113.215.179。

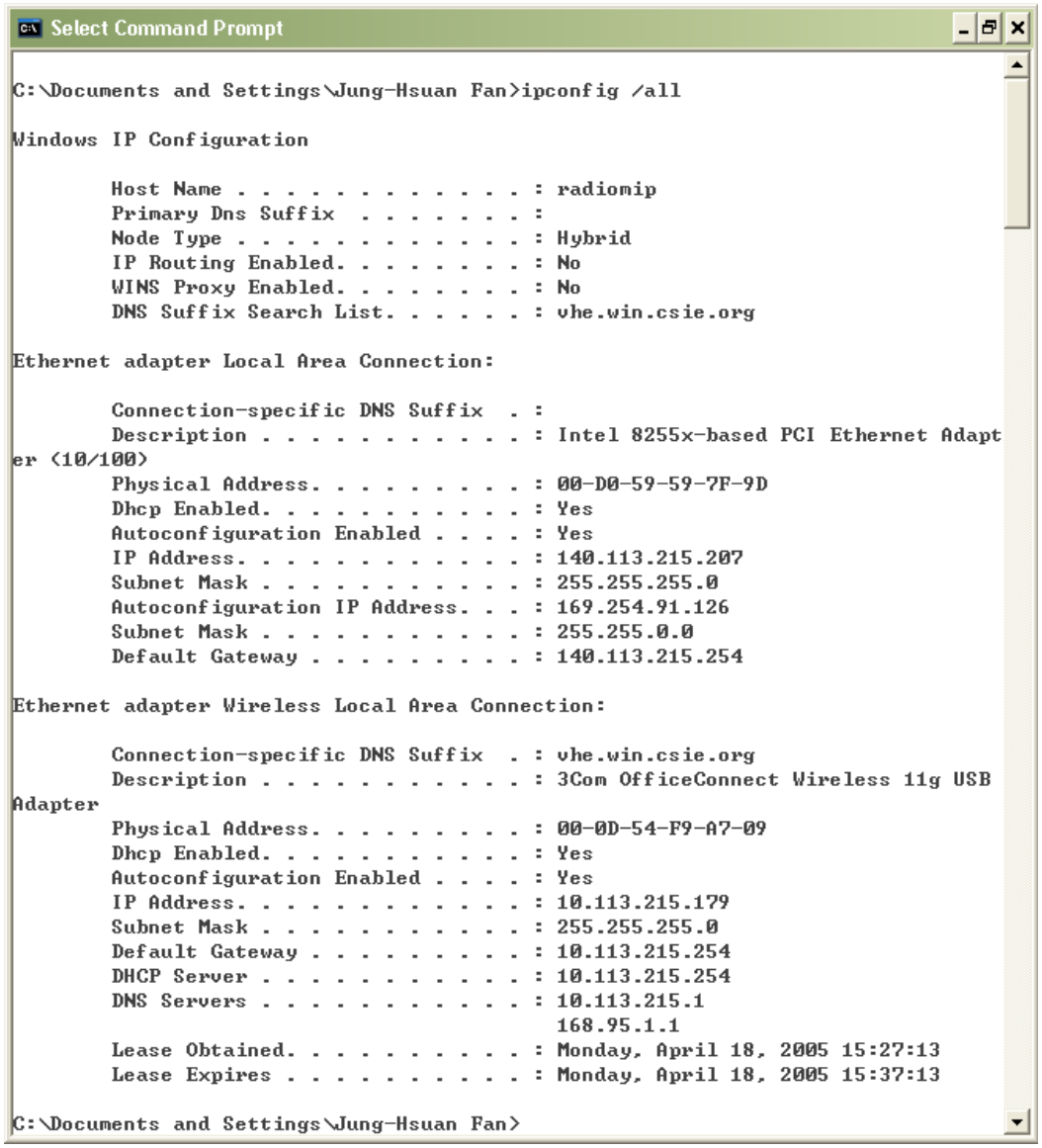

Figure 6 - 12: Mobile Node IP Address Configuration

雖然 IP 別名的技巧可以讓主配接卡擁有家位址的同時也能透過 DHCP 協定取得配置轉交位址，卻存在一個潛在的問題。 當主配接卡的網路線拔除後，媒體感應事件會觸發 TCP/IP 協定堆疊移除該網路卡所使用的 IP 位址與相關路由。 如此一來將造成行動端家位址的遺失，而原本使用該位址的應用程式也被迫更換，在此情況下，連線便無法維持。 因此，基於這個原因，我們才使用 Mobile IP 服務中間層驅動程式來保持家位址的存在。

- 路由表與 ARP 快取之管理 (Routing Table and ARP Cache Magamenent)

當系統同時擁有多張運作中的網路卡時就意味著我們在路由表中會看到多筆的相關路由設定。 然而為了讓應用程式不管在家網域或是外地網域下皆使用家位址作為送出封包的來源 IP 位址，意味著我們勢必要對路由表做一些適當的調整，以下便透過實際的路由表詳細的解說修改的內容：

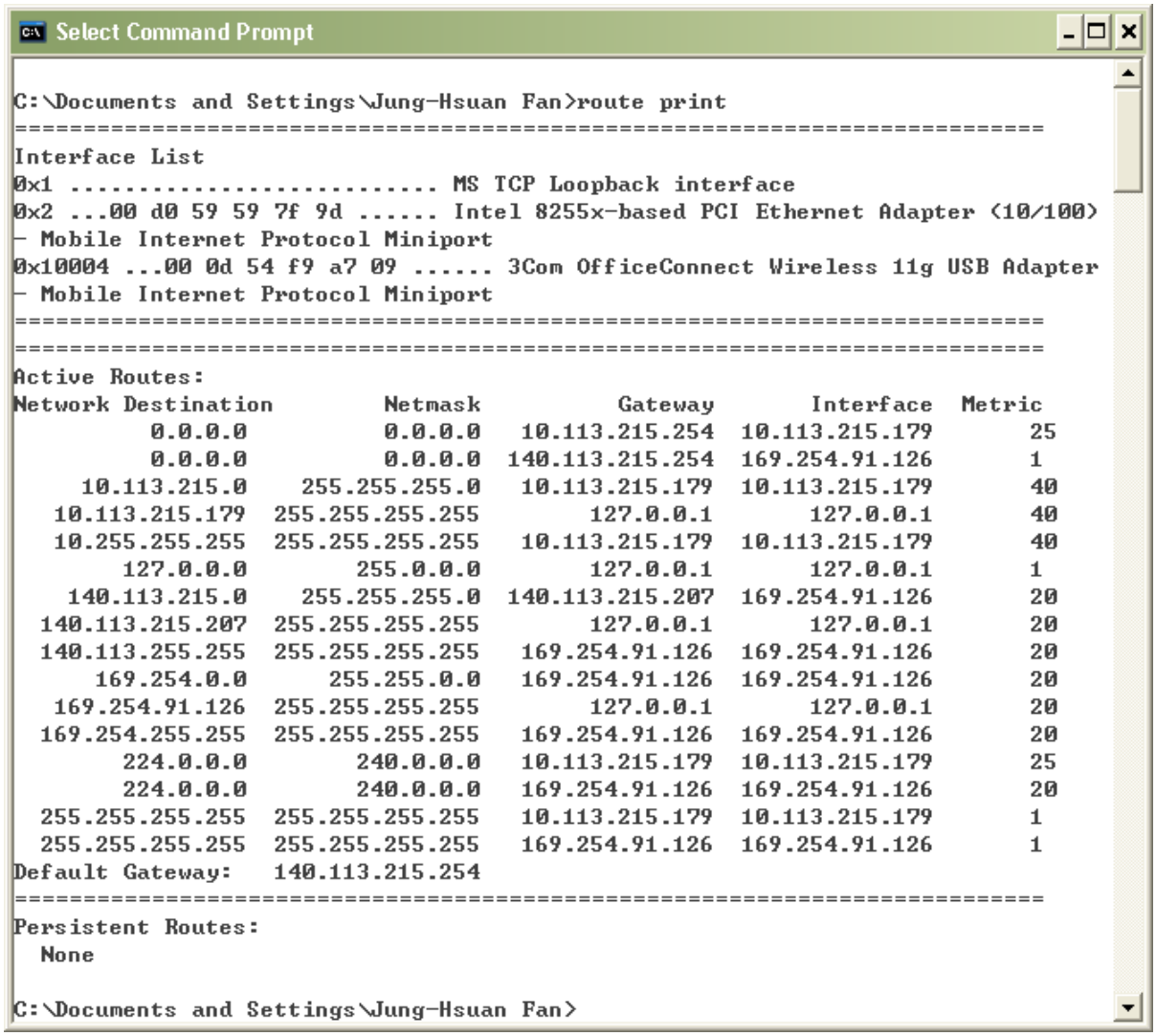

Figure 6 - 13: Routing Table under Home Network

首先，圖 6-13 為行動端在家網域時的路由表資訊。 由於主副配接卡在不同的網路下，所以會有其各自的 IP 位址與閘道器之設定，而在路由表上所反映的是存在多個預設路由。 這情況便是在 2.4.2 所提到的多重網路連線，也就是系統會選擇一預設路由作為明確的預設路由，如圖下方的預設閘道(Default Gateway)所示。 由於 TCP/IP 協定堆疊會根據該預設閘道填入封包適當的來源位址，以下圖為例，當預設閘道為 140.113.215.254，則封包的來源位址便會是與它同網域的網路卡 IP 位址 140.113.215.207，因此為了讓應用程式的封包皆以家位址作為來源 IP 位址，我們必須讓預設閘道永遠為家網域的閘道器或是某台主機。 基於這個原因我們便將擁有家位址的主配接卡之預設路由公制設為 1，也就是獲得最高的優先權。 如此一來，系統便會選擇其作為預設閘道。

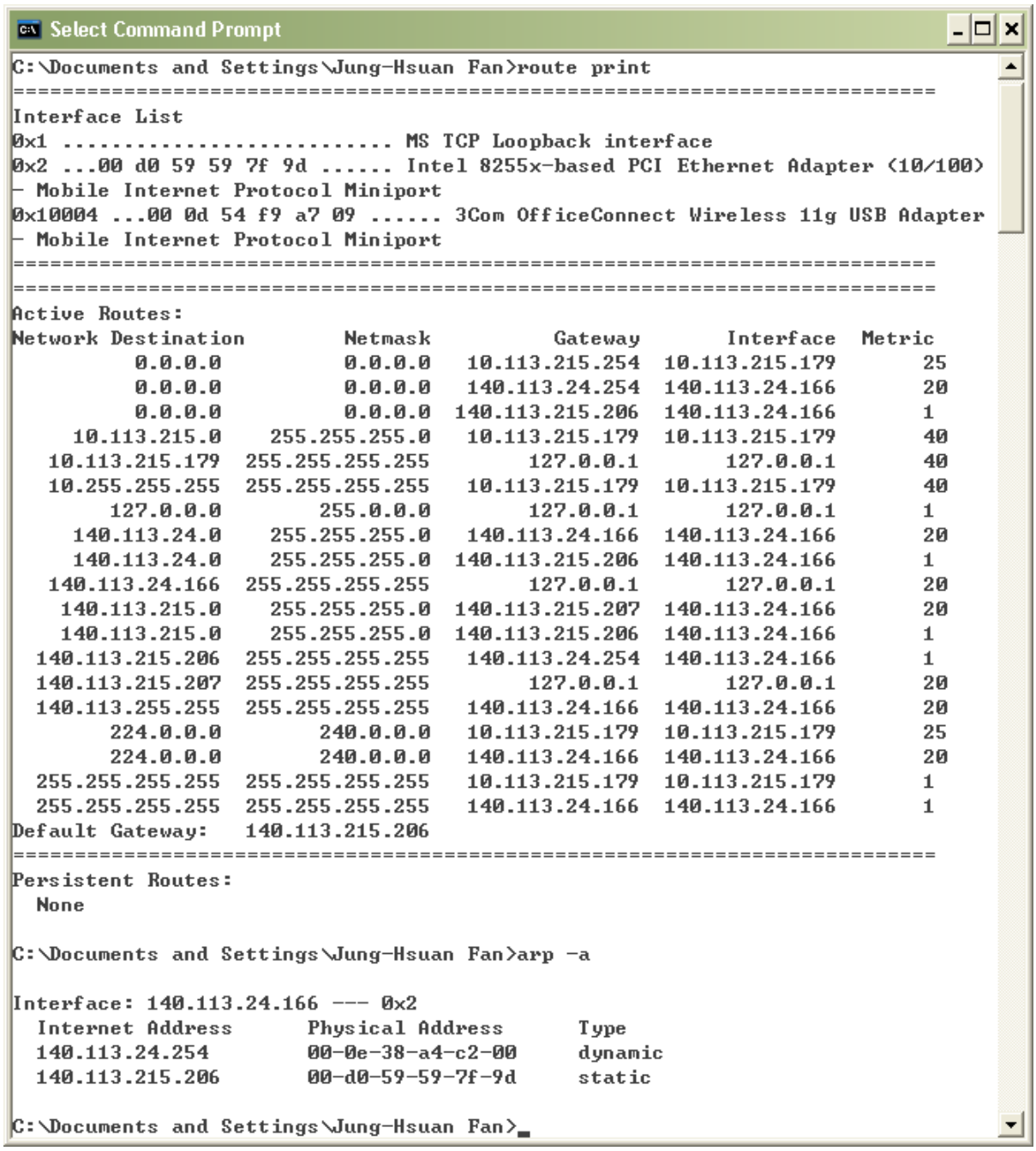

Figure 6 - 14: Routing Table under Foreign Network (LAN)

接著，行動端漫遊到外地網域，主配接卡透過 DHCP 協定取得當地所使用的 IP 位址 140.113.24.166 與閘道器位址 140.113.24.254 後，系統會在路由表中新增相對應的路由，如圖 6-14 所示。 然而同樣為了讓應用程式能夠繼續使用家位址來維持原有的連線，我們必須對路由表做些許的調整，以下分成三個部分做解釋：

1. **預設路由 (Default Route)**。 在我們的實驗環境中所有的路由器皆具備 2.1.4 所提的進出過濾器，所以必須採用反向通道技術來克服，也就是當行動端在外地網域時封包必須先經過封裝後送回家代理器。 因此，我們在這裡便將家代理器 140.113.215.206 設為預設閘道器，方法如同前述，即是將其對應的預設路由公制設為 1。 接著根據

2.4.3 的描述，當封包欲遞送到家代理器之前，TCP/IP 協定堆疊會以 ARP 封包執行實體位址的解析。 然而，根據行動網際網路協定的規範，在外地網域是禁止以家位址發送任何的 ARP 封包。 為了符合 RFC 的標準，所以我們在系統的 ARP 快取中增加了對於家代理器的靜態設定。 此處技巧的作法是將其 MAC 位址設為與主配接卡相同實體位址。

2. **直接連線的網路 (Directly Attached Network)。** 對於直接連線的網路來說，會以當地的 IP 位址作為來源位址。 以上圖為例，原本 140.113.24.0/24 的閘道器(即次躍點)為 140.113.24.254，則系統會以與之同網域的 IP 位址 140.113.24.166 作為封包的來源位址。 然而，這樣便違背了行動網際網路協定的規範。 所以我們必須以公制為 1 且閘道器為家代理器的路由來取代之，因此我們新增了 140.113.24.0/24 及 140.113.215.0/24 兩個直接連線的網路之取代路由。
3. **本地主機 (Local Host)。** 至於要送到家代理器的封包如註冊請求訊息，必須以當地所取得的 IP 位址，也就是所謂的配置轉交位址作為封包的來源 IP 位址。 因此我們新增了一項到家代理器 140.113.215.206 的本地主機路由，其中閘道器為外地網域的閘道器 IP 位址 140.113.24.254 而公制為 1。

當切換到同樣位在外地網域的副配接卡，此時的動作就如同上述的步驟，如圖 6-15 所示。 不過在步驟二必須稍加留意，由於副配接卡所處的為私有網域 10.113.215.0/24，如果如同原本的步驟將送往該網域的封包經過封裝送回家代理器會造成封包被丟棄(無法遞送目的位址為私有網際網路位址)。 因此，假如所得到的配置轉交位址為一私有 IP 位址，則跳過步驟二中對於該路由的動作。 意即我們將以當地的 IP 位址 10.113.215.179 直接與 10.113.215.0/24 網域下的其他主機通訊，不過如此一來便無法保障網路切換時的連線維持，但此問題不在本論文所探討的範疇內。 此外與圖 6-14 相異的地方就是在於步驟三對家代理器的路由，此處我們可以看到該閘道器設定已由原本主配接卡的 140.113.24.254 換為副配接卡的 10.113.215.254。 也就是說，將透過副配接卡所取得的配置轉交位址來與家代理器註冊，如此便可完成不斷線的網路卡切換。

最後，當行動端回到家網域時，路由表便修改回如圖 6-13 的狀況。 以上便是在異質網路間漫遊的過程中所有路由表的變化情形。

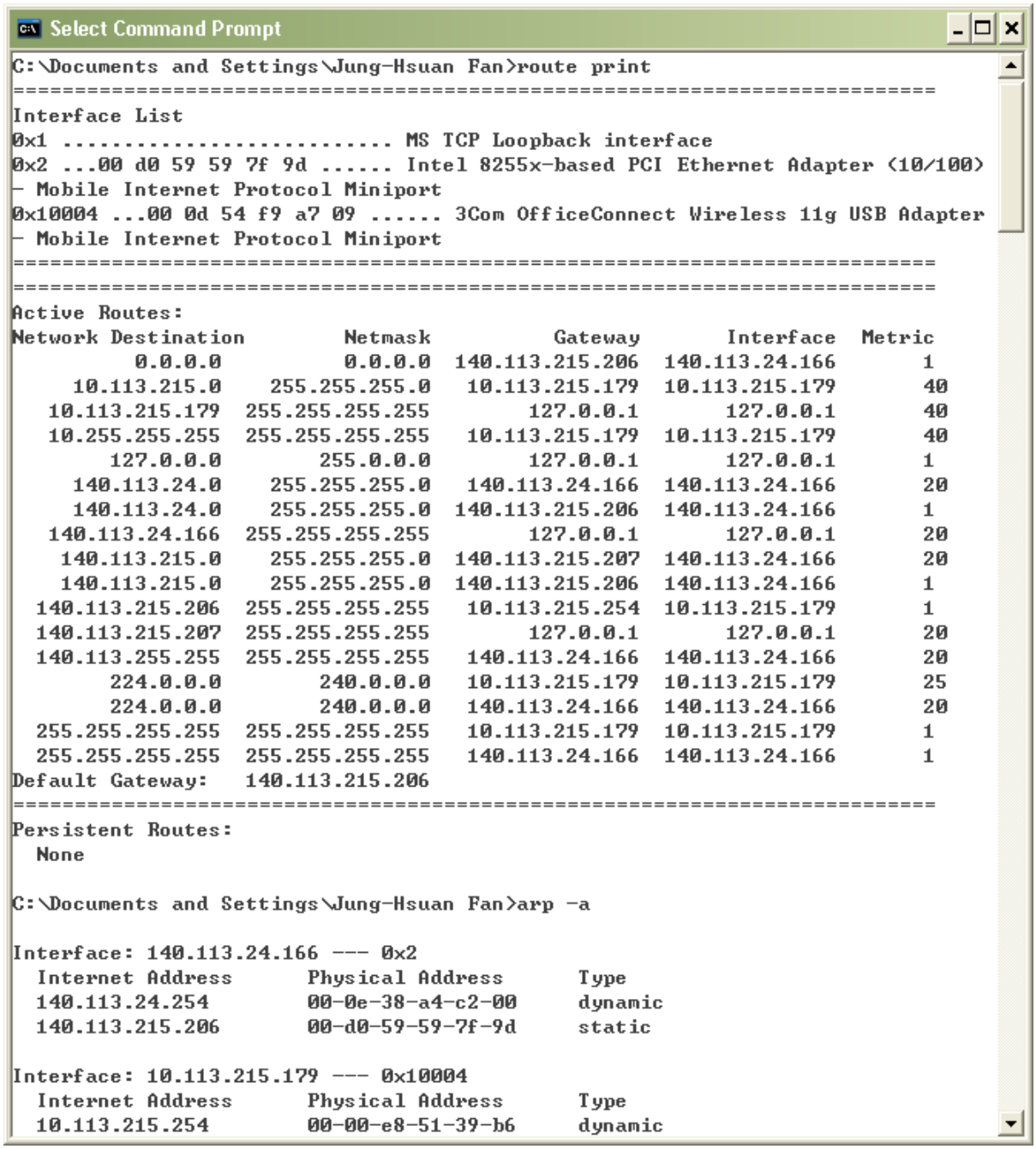

Figure 6 - 15: Routing Table under Foreign Network (WLAN)

- 切換之抉擇 (Handoff Decision)

在網路卡使用的切換上，我們提供了充足的資訊，例如訊號強度、存取點列表與媒體連結狀態等讓使用者開發各式各樣的抉擇演算法。 在此我們僅展示一種以優先權為考量的切換抉擇演算法，其精神為優先使用主配接卡，唯有當主配接卡無法使用時才切換到副配接卡，而一旦主配接卡恢復則立即切換之。 圖 6-16 便是詳細的演算法流程圖，一開始將主配接卡設為運作配接卡後便等待事件的發生。 第一種事件為媒體狀態改變指示，也就是發生在網路卡連線中斷的時候，當接收到此事件後會重新的選擇運作配接卡。 另一種事件則為 DHCP 協定所觸發的，也就是當取得新的 IP 位址。 不管上述何種

事件發生，最後的步驟皆為執行運作配接卡的更新動作(包含更新相關路由及行動網際網路協定的重新註冊等)。

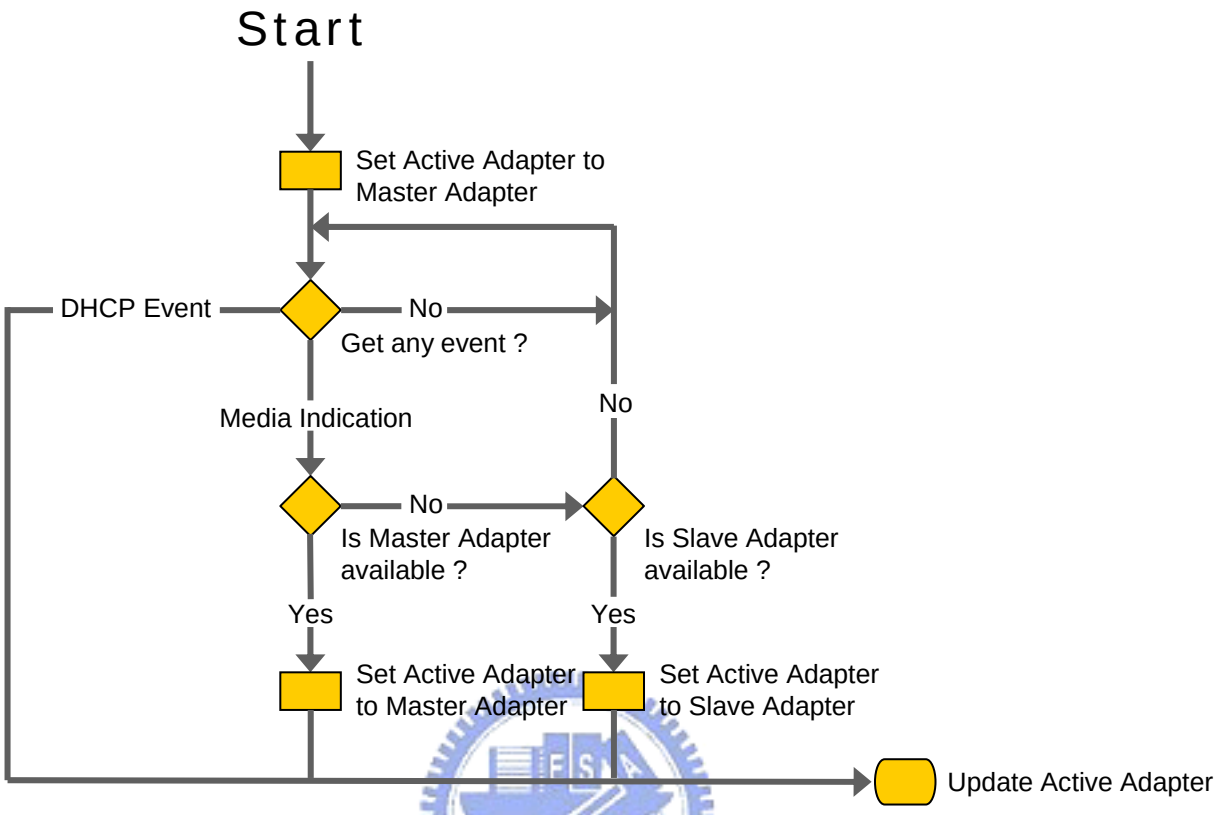

Figure 6 - 16: Handoff Decision Algorithm

- 切換之處理 (Handoff Process)

網路卡切換的第一個步驟為根據運作配接卡目前的 IP 位址判斷行動端目前所處的行動網際網路協定狀態，詳細的流程圖如下所示。 接著，根據不同的狀態會有不同的後續處理動作，以下便分別解釋圖中四個狀態所代表的意義與其對應的處理程序：

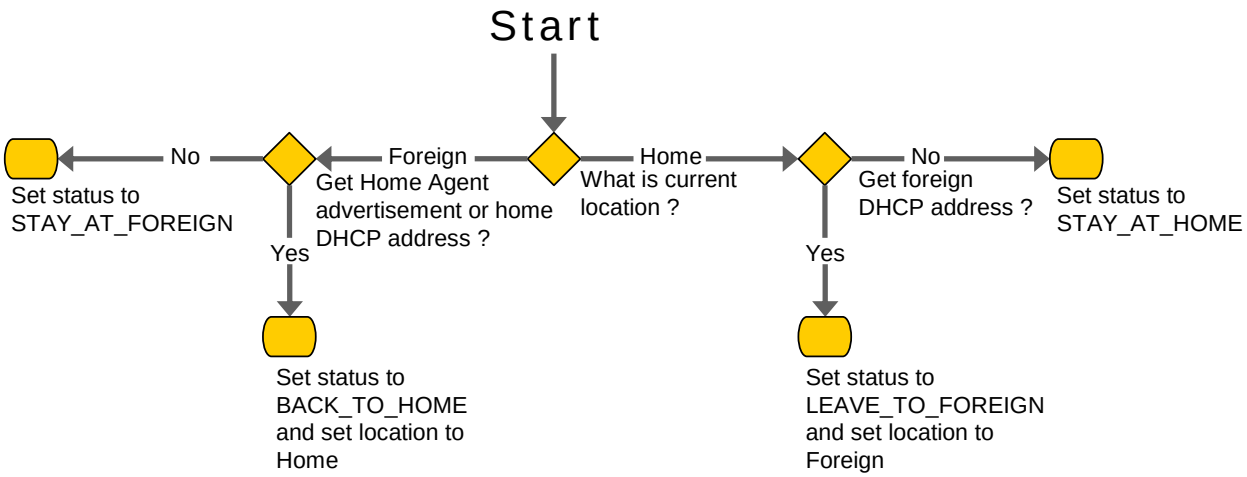

Figure 6 - 17: Mobile IP Status Determination Algorithm

1. **STAY_AT_HOME。** 這個狀態表示行動端仍在家網域內，因此無須額外的動作。
2. **LEAVE_TO_FOREIGN。** 這個狀態表示行動端剛由家網域移至外地網域。 首先以前述的 IPFLT_SET_IPADDR 與 IPFLT_FLT_IP 命令通知 IpFlt 驅動程式攔截所有來源 IP 位址為家位址的封包以進行封裝，並以 IPFLT_FLT_ARP 告知 IpFlt 驅動程式阻擋以家位址所發送的 ARP 封包。 接著根據上述路由表與 ARP 快取之管理原則修改路由表與 ARP 快取。 最後以運作配接卡透過 DHCP 所取得的 IP 位址作為配置轉交位址向家代理器送出 Mobile IP 的註冊要求訊息。
3. **STAY_AT_FOREIGN。** 這個狀態表示行動端仍停留在外地網域，只是運作配接卡可能由主配接卡換為副配接卡或是相反的情形。 因此，這邊的處理動作就只是根據上述路由表與 ARP 快取之管理原則修改路由表與 ARP 快取後再以運作配接卡透過 DHCP 所取得的 IP 位址作為配置轉交位址向家代理器送出 Mobile IP 的註冊要求訊息。
4. **BACK_TO_HOME。** 這個狀態表示行動端剛由外地網域回到家網域。 因此，我們必須用先前的 IPFLT_FLT_IP 與 IPFLT_FLT_ARP 命令通知 IpFlt 驅動程式解除攔截特定 IP 與 ARP 封包的動作。 接著根據上述路由表與 ARP 快取之管理原則修改路由表與 ARP 快取。 最後向家代理器送出 Mobile IP 的解除註冊要求訊息。

6.1.3 Software Demonstration

在本節的最後我們將以實際的程式展示兩個應用實例，首先是 4.3.1 有線區域網路與無線區域網路間的漫遊情境，並用 FTP 傳輸做為網路切換時不斷線的證明。

- Software Configuration
  - 家代理器 (Home Agent)

    在家代理器的部分必須先載入 IP-in-UDP 的封裝模組後再啟動家代理器的主程式，如圖 6-18 所示。

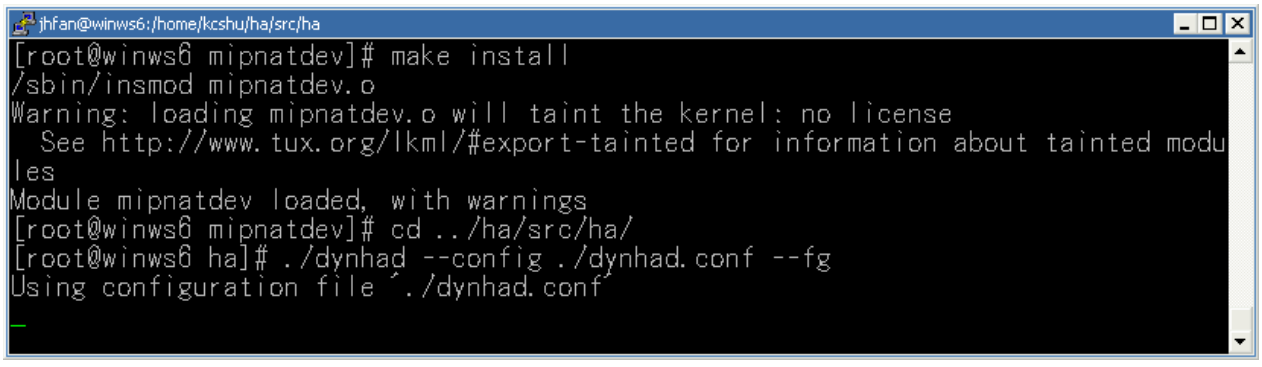

Figure 6 - 18: Home Agent Configuration Snapshot

- 行動端 (Mobile Node)

執行 RIOMIP 程式後首先會出現設定視窗。 在配接卡的部分，我們選擇有線的 Intel 區域網路卡作為主配接卡，而副配接卡便選 3Com 的無線區域網路卡。 而在行動網際網路協定相關的選項方面則必須填入家位址與家代理器等資訊，詳見圖 6-19。

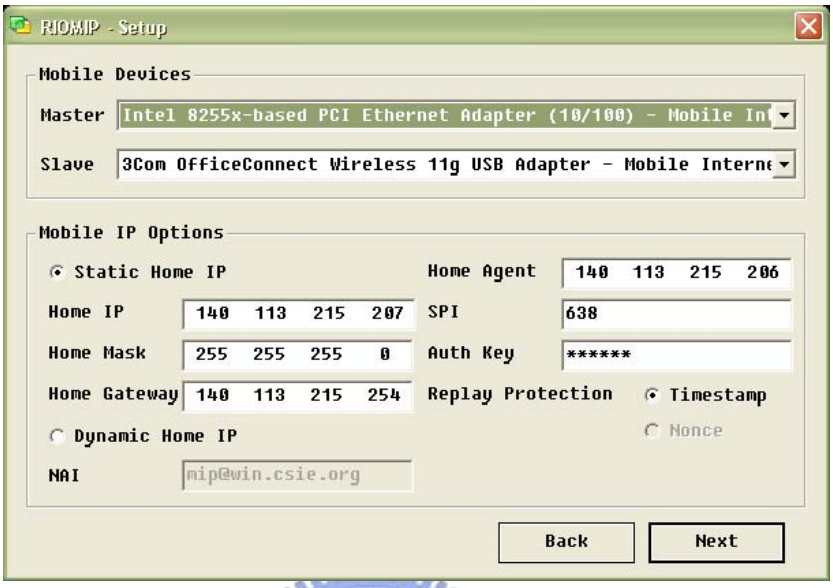

Figure 6 - 19: Mobile Node Configuration (LAN/WLAN) Snapshot

Program Snapshots

啟動系統的運作後，透過事件顯示視窗(見圖 6-20)我們可以看到行動端已送出註冊要求訊息並且收到表示成功的回覆。 在下方的資訊欄中，顯示著目前所使用的網路卡，配置轉接位址以及封裝方式等資訊供使用者參考。 同時，為了測試不斷線的網路卡切換，我們開啟 FTP 應用程式 CuteFTP 並連線到通訊端 linux.sinica.edu.tw (140.109.13.40)下載新版的 Linux 作業系統 fedora，此時我們可以看到傳輸速度為每秒 1.34MB。

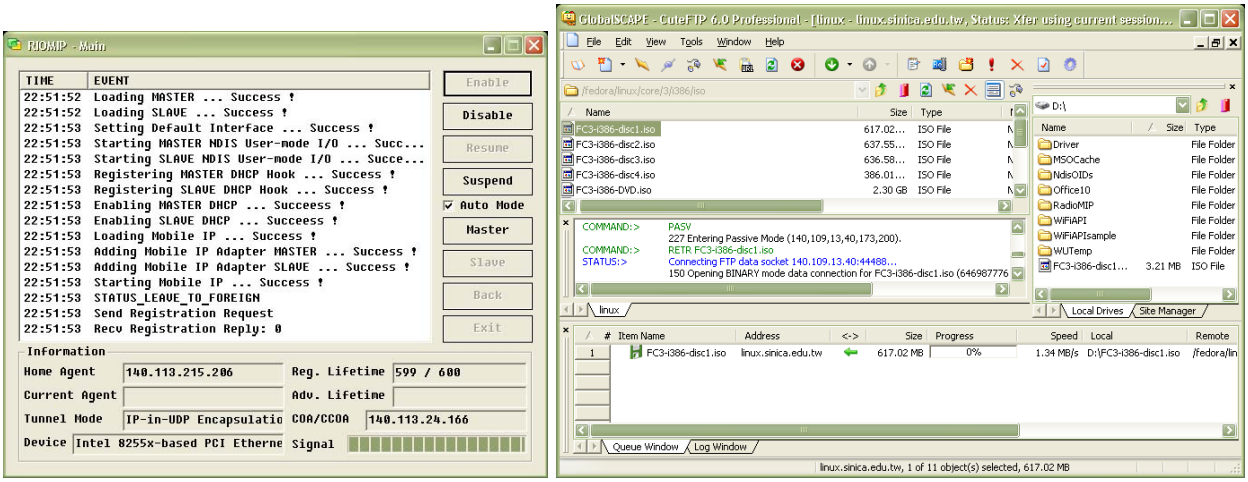

Figure 6 - 20: Mobile Node Initial with LAN Adapter Snapshot

接著，我們拔除 Intel 主配接卡上的網路線，如圖 6-21 所示。 此時我們看到 RIOMIP 程式立即將網路卡換為 3Com 副配接卡，並且送出新的註冊要求訊息。 此外，下方的訊息欄位所顯示的配置轉接位址也已更新，並且可以看到無線區域網路卡的訊號強度。 而在 CuteFTP 程式這邊我們發現到連線並沒有中斷，唯一有改變的是傳輸速度降為每秒 193KB。

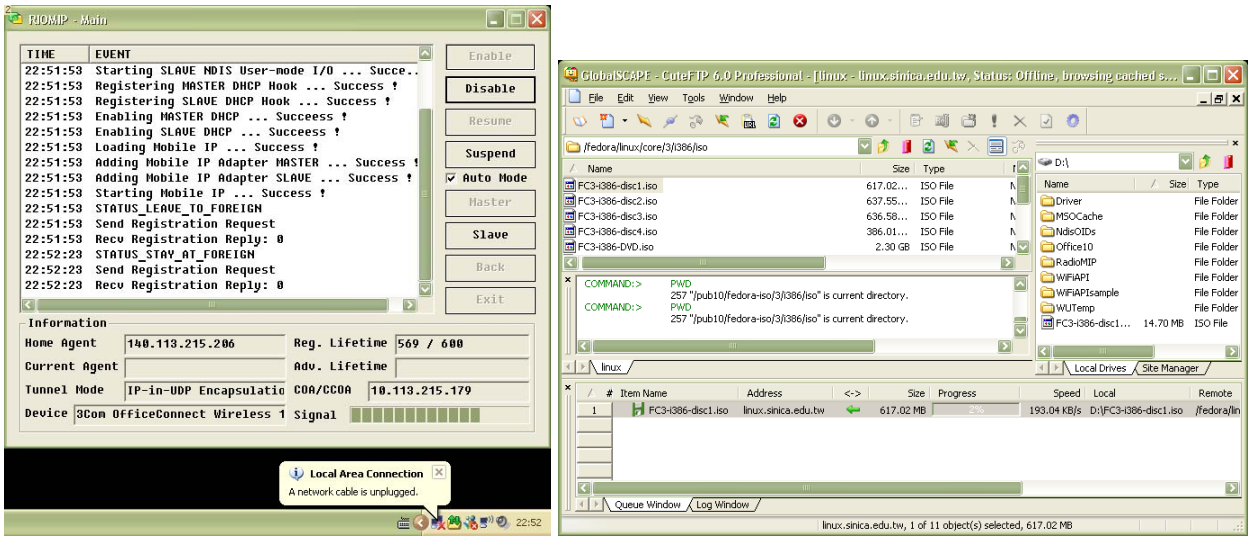

Figure 6 - 21: Mobile Node Handoff to WLAN Adapter Snapshot

最後，我們將主配接卡的網路線再度接上。 根據前述的網路卡切換抉擇演算法，RIOMIP 程式立自動即換回使用主配接卡。 同樣我們可以在程式的畫面中(如圖 6-22 所示)看到這樣的變化，此時再度檢視 FTP 應用程式的視窗可以發現傳輸速度又回到每秒 0.99MB。

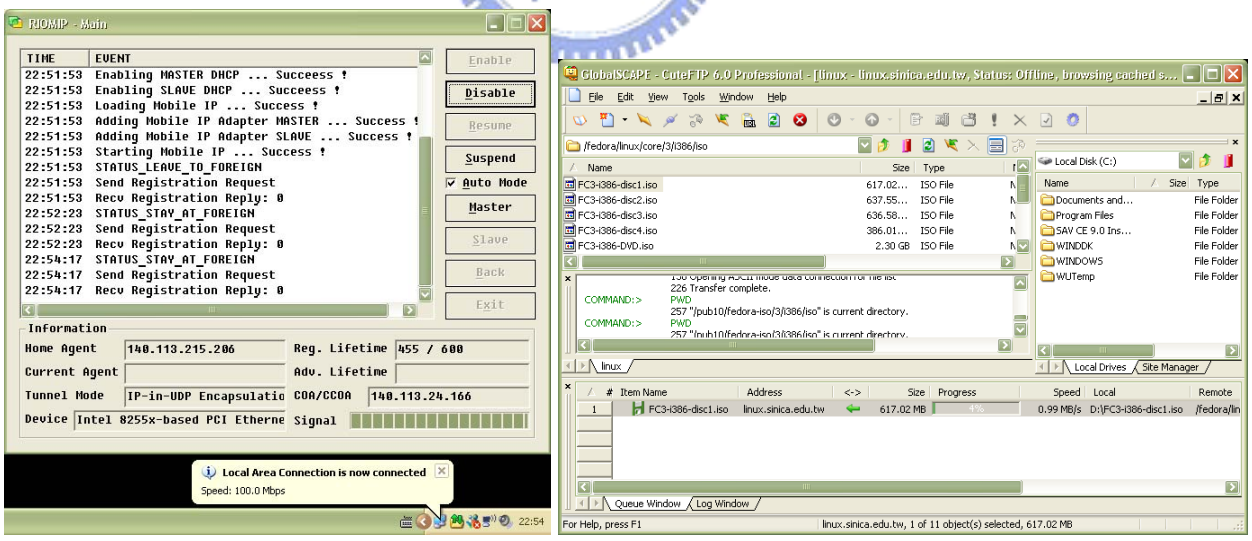

Figure 6 - 22: Mobile Node Handoff to LAN Adapter Snapshot

接下來讓我們再看看 4.3.2 無線區域網路與整合封包無線電服務的漫遊情境，這次我們將以 ICMP 回應請求訊息作為展示實例。

- Software Configuration
  - 家代理器 (Home Agent)

    在家代理器的部分由於如同上個實例的設定，在此便不再累述。
  - 行動端 (Mobile Node)

    在這個實例中，我們選擇 3Com 的無線區域網路卡作為主配接卡，而副配接卡則選 WAN 介面卡，也就是 GPRS 撥號配接卡所對應的裝置。

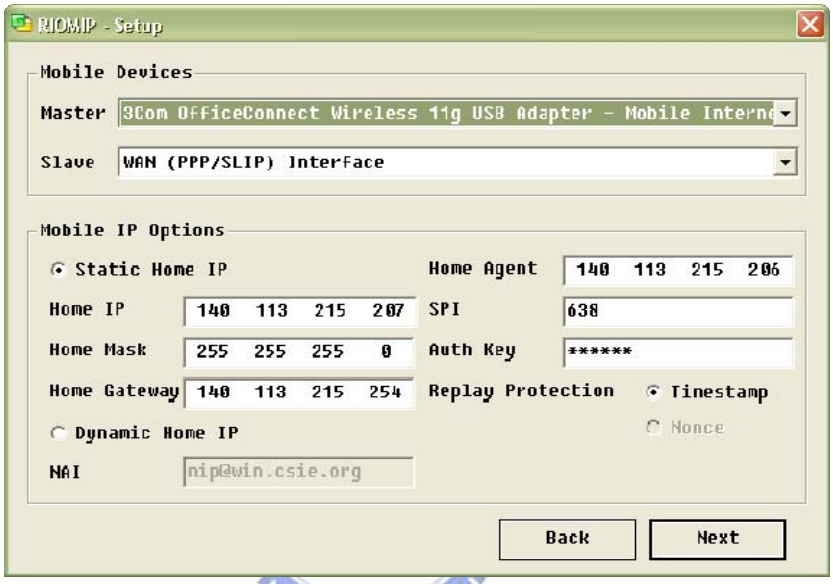

Figure 6 - 23: Mobile Node Configuration (WLAN/GPRS) Snapshot

- Program Snapshots

設定完成後便如同前述的方法啟動系統的運作，不過這次在測試上我們改採持續向通訊端 linux.sinica.edu.tw (140.109.13.40) 以微軟視窗作業系統內建的工具程式 ping 發送 ICMP 的回應請求訊息(如圖 6-24 所示)，藉以觀察在稍後執行切換的動作時是否會發生封包的遺失。

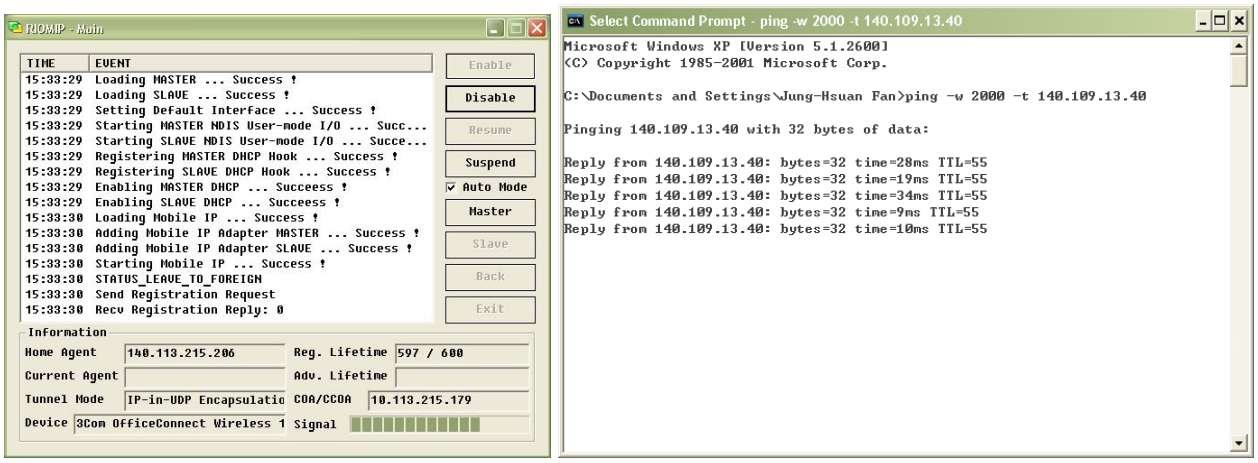

Figure 6 - 24: Mobile Node Initial with WLAN Adapter Snapshot

接著，我們按下手動切換副配接卡的按鈕，如圖 6-25 所示。 此時我們看到 RIOMIP 程式立即將使用網路卡換為標示 WAN 介面之 GPRS 撥號配接卡，並且送出新的註冊要求訊息。 同時下方的訊息欄位也顯示著更新過後的配置轉接位址。 而在 ping 程式這邊我們發現到原本以 WLAN 送出的回應要求訊息之來回時間(Round Trip Time, RTT)大約皆小於 50ms，但在切換到 GPRS 後，來回時間增加到了 1000ms 左右。 不過在切換的過程中並沒有發生封包的遺失，如同上個例子，僅是造成速度的下降。

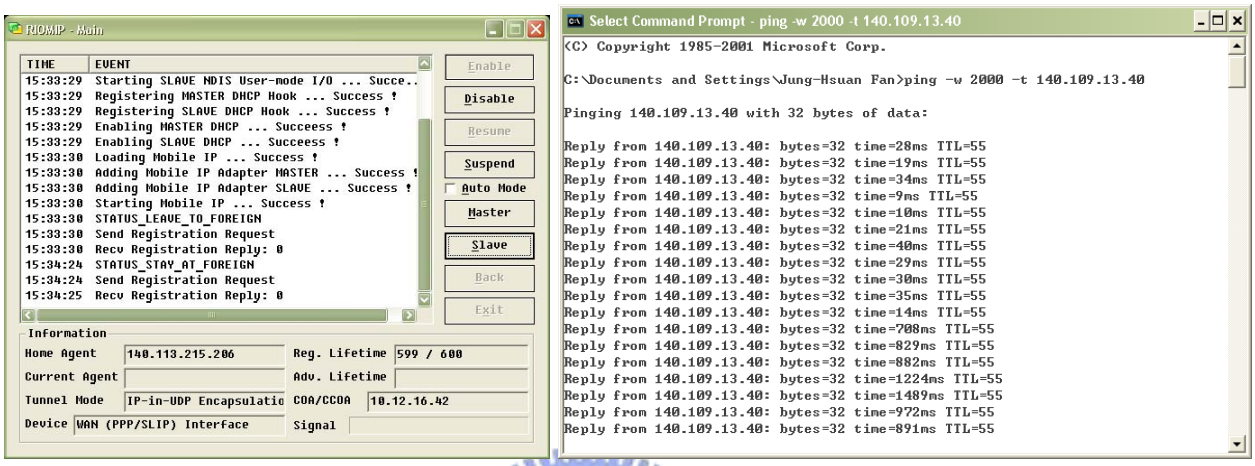

Figure 6 - 25: Mobile Node Handoff to GPRS Adapter Snapshot

最後，我們再度以手動的方式切換回主配接卡，同樣地可以在程式的畫面中(如圖 6-26 所示)看到這樣的變化，此時再度檢視 ping 程式的視窗可以發現來回時間又縮減到小於 50ms。 經過統計發現，在以上的切換過程中，我們一共送出 35 個回應要求訊息，也收到 35 個回應回覆訊息，即表示我們沒有造成任何一個封包的遺失。

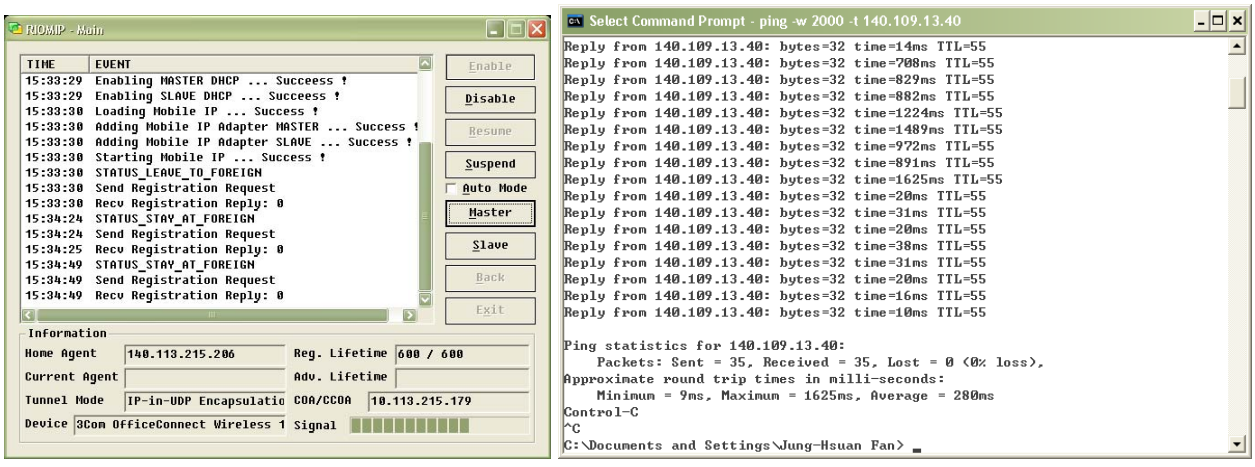

Figure 6 - 26: Mobile Node Handoff to WLAN Adapter Snapshot

Section 6.2 Implementation of Alternative Tunneling

本節將接著介紹如何實作第五章所提及的 IP^{HO} 承載 IP 之封裝模式與網路位址交換模組。 首先在 6.2.1 會說明實作上所使用的軟硬體設備及網路拓樸。 接著在 6.2.2 便分別介紹如何在家代理器與網路位址轉譯器上新增我們所需的軟體元件。 最後 6.2.3 的部分則為程式的操作實例。

6.2.1 Environment Configuration

在這部分的實作環境主要包含三個主要的端點，分別是家代理器，行動端與網路位址轉譯器。 以下便針對其上所需使用的軟硬體及程式做一簡單的介紹後，再說明整體環境的網路設定。

- 家代理器 (Home Agent)

  家代理器所使用的機器為一普通的桌上型電腦並配有乙張乙太網路卡，其詳細的硬體規格如下表所列：

Category	Specification
PC	Desktop
Processor	Intel Pentium III (Coppermine) 933 Mhz
Memory	Kingstone PC-133 SDRAM 512 MB
HDD	Seagate ST380021A 7200RPM 80GB
Ethernet	Intel 8255x-based PCI 10/100Mbps

Table 6 - 7: Home Agent Hardware Requirement Table

- 行動端 (Mobile Node)

  行動端所使用的機器為一筆記型電腦搭配有內建的乙太網路卡，詳細的硬體規格如下表所列：

Category	Specification
PC	Acer TravelMate 273X Notebook
Processor	Intel Pentium 4-M 1700 Mhz
Memory	Kingstone DDR 266 SDRAM 384 MB
HDD	IBM IC25N020ATCS04-0 4200RPM 20GB
Ethernet	RealTek RTL8139 PCI 10/100Mbps

Table 6 - 8: Mobile Node Hardware Requirement Table

- 網路位址轉譯器 (Network Address Translator)

  網路位址轉譯器所使用的機器為一普通的桌上型電腦並配有兩張乙太網路卡，詳細的硬體規格如下表所列：

Category	Specification
PC	Desktop
Processor	AMD Athlon 1000 Mhz
Memory	Kingstone PC-133 SDRAM 256 MB
HDD	Maxtor 5T040H4 7200RPM 40GB
Ethernet	RealTek RTL8139 PCI 10/100Mbps
	Digital DS21143 PCI 10/100Mbps

Table 6 - 9: Netwok Address Translator Hardware Requirement Table

■ 軟體需求 (Software Requirement)

- 家代理器 (Home Agent)

  家代理器所使用的軟體如下表所列，其運行的作業系統為 Redhat 版本 9.0 的 Linux，而執行的家代理器程式為支援 IP^{HO} 承載 IP 封裝方式的 Dynamics Mobile IP 套件之修改版本。

Category	Specification
Operating System	Redhat Linux 9.0 Kernel 2.4.18-14
Mobile IP Package	HUT Dynamics 0.8.1*
IP-in-IP^{HO} Tunnel Device	ipip.o*

Table 6 - 10: Home Agent Software Requirement Table

- 行動端 (Mobile Node)

  行動端所使用的軟體如下表所列，其運行的作業系統為 Fedora 版本 2 的 Linux，而執行的家代理器程式為原始由芬蘭赫爾辛基科技大學所開發的 Dynamics 版本 0.8.1 之 Mobile IP 套件。

Category	Specification
Operating System	Fedora Core 2 Kernel 2.6.5-1.358
Mobile IP Package	HUT Dynamics 0.8.1
IP-in-IP Tunnel Device	ipip.o

Table 6 - 11: Home Agent Software Requirement Table

- 網路位址轉譯器 (Network Address Translator)

網際網路位址轉譯器所使用的軟體如下表所列，其運行的作業系統為 Fedora 版本 2 的 Linux，而執行的網路位址轉譯模組則為修改自 Netfilter 所發展的 iptables 套件。

Category	Specification
Operating System	Redhat Linux 9.0 Kernel 2.4.26
NAT Module	iptables 1.2.7a*

Table 6 - 12: Home Agent Software Requirement Table

■ 網路拓撲 (Network Topology)

圖 6-23 所展示的便是我們在以下實作過程中所使用的網路環境設定。 其中家網域及外地網域的區域網路，也就是網路前置碼為 140.113.0.0 之網路為交通大學所屬。 而通訊端所在的網域 140.109.0.0，則是屬於中央研究院所有。

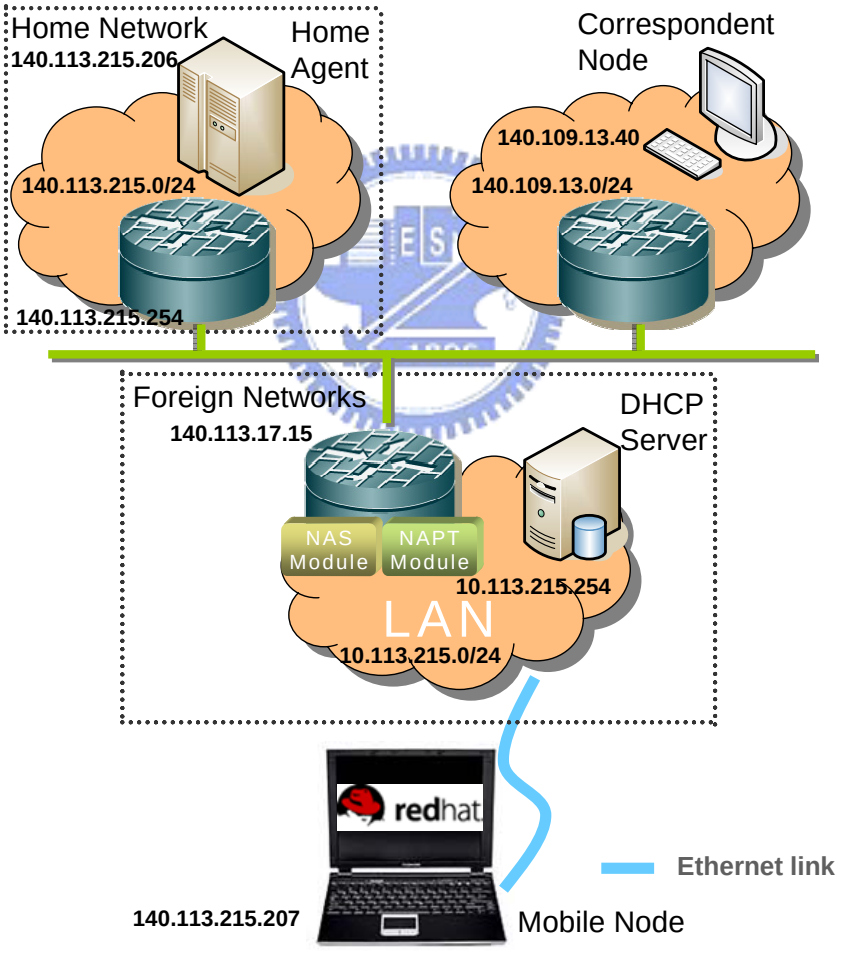

Figure 6 - 27: Alternative Tunneling Implementation Network Topology

6.2.2 Software Development

以下為實作家代理器內的 IP^{HO} 承載 IP 通道模組與網路位址轉譯器上的網路位址交換功能之解說：

- IP-in-IP^{HO} Tunneling Module

IP^{HO} 承載 IP 通道模組的製作是修改原有系統內的 IP 承載 IP 通道模組 ipip.o，我們在其內新增一個裝置控制命令讓它可以判斷是否在封裝外層的 IP 標頭時加入型態 252 的標頭選項。 以下便列出詳細的修改部分：

- 資料結構 (Data Structure)

首先針對型態 252 的標頭選項，我們必須在系統中新增其資料結構以供通道模組來使用。 因此，在核心原始碼的路徑/usr/src/linux/include/linux/netfilter_ipv4 下，我們加入了定義圖 6-24 之資料結構的檔案 ip_nas.h。

```
struct ip_nas_opt {
        unsigned char code;
        unsigned char length;
        unsigned char count;
        unsigned char reserved;
        unsigned long addr;
};
```

Figure 6 - 28: IP Header Option Data Structure Definition

- 裝置控制 (Device I/O Control)

接著我們修該原始碼路徑/usr/src/linux/include 下的檔案 if_tunnel.h，目的是在原始的通道模組中增加一新的 I/O 命令，其名稱與用途如下表左列：

IOCTL Code	Description
SIOCSETTUNNEL	啟用型態 252 之標頭選項並設定內所要放置的 IP 位址

Table 6 - 13: IP-in-IP^{HO} Tunneling Device I/O Control Table

- 通道裝置 (Tunneling Device)

最後我們修改原始碼路徑/usr/src/linux/net/ipv4 下的檔案 ipip.c，該檔案便是原始的 IP 承載 IP 通道模組。 第一個步驟為修改該通道裝置在封裝封包時所會讀取的參數，如圖 6-25 所示，在原有的參數最後我們加入了變數 nas_addr 用來存放型態 252 標頭選項內的 IP 位址。 也就是當用上述的裝置控制命令 SIOCSETTUNNEL 所傳入的位址將會被放置在該變數內。

```
struct ip_tunnel_parm
{
        char            name[IFNAMSIZ];
        int             link;
        __u16           i_flags;
        __u16           o_flags;
        __u32           i_key;
        __u32           o_key;
        struct iphdr    iph;
        __u32           nas_addr;
};
```

Figure 6 - 29: Tunneling Device Parameter

再來就是修改實際執行封包封裝的函式 ipip_tunnel_xmit，圖 6-26 列出了該函式中重要的部分。 在該函式的處理中，首先檢查上述參數中的 nas_add 變數是否已存入 IP 位址，若有則預留標頭選項所需的額外空間。 接著在封裝外層標頭的部分也先做同樣的檢查，若 nas_add 變數已存入 IP 位址，則緊接著外層的 IP 標頭填入上述的資料結構 ip_nas_opt，也就是型態 252 之標頭選項。 並且把 nas_add 變數內的 IP 位址填在該結構的交換位址欄位 addr。

```
static int ipip_tunnel_xmit(struct sk_buff *skb, struct net_device *dev)
{
.....
        if(tunnel->parms.nas_addr != INADDR_ANY)
                skb->nh.raw = skb_push(skb, sizeof(struct iphdr) + sizeof(struct ip_nas_opt));
        else
                skb->nh.raw = skb_push(skb, sizeof(struct iphdr));
        memset(&(IPCB(skb)->opt), 0, sizeof(IPCB(skb)->opt));
        dst_release(skb->dst);
        skb->dst = &rt->u.dst;

        /*
         *      Push down and install the IPIP header.
         */

        iph             =       skb->nh.iph;
        iph->version    =       4;
        iph->ihl        =       sizeof(struct iphdr)>>2;
        iph->frag_off   =       df;
        iph->protocol   =       IPPROTO_IPIP;
        iph->tos        =       INET_ECN_encapsulate(tos, old_iph->tos);
        iph->daddr      =       rt->rt_dst;
        iph->saddr      =       rt->rt_src;
        skb->ip_summed = CHECKSUM_NONE;
        if(tunnel->parms.nas_addr != INADDR_ANY){
                struct ip_nas_opt *nasopt = (struct ip_nas_opt *)((unsigned char *)iph + sizeof(struct iphdr));

                iph->ihl += sizeof(struct ip_nas_opt) >> 2;
                nasopt->code = 252;
                nasopt->length = 8;
                nasopt->count = 0;
                nasopt->reserved = 0;
                nasopt->addr = tunnel->parms.nas_addr;
        }
.....
}
```

Figure 6 - 30: Tunneling Device Internal Function

- HUT Dynamics Mobile IP Package Modification

接下來必須修改 Dynamics 套件讓它可以使用我們所開發的 IP^{HO} 承載 IP 通道模組。 在這部分一共修改了套件中的三個檔案，以下分成兩部分說明：

- 通道裝置控制 (Tunneling Device Control)

我們在原始碼路徑/dynmacis/src/other 下的檔案 dyn_ip.c 內增加了用來建立 IP^{HO} 承載 IP 通道的函式 dyn_ip_tunnel_add_nas，如下圖所示。 在該函式內所呼叫的 tunnel_ioctl 函式中便以我們所新增的裝置控制命令 SIOCSETTUNNEL 設定型態 252 之標頭選項內所要填入的 IP 位址。

```
int dyn_ip_tunnel_add_nas(const char *dev, struct in_addr remote,
                  struct in_addr local, int nas_addr)
{
      return tunnel_ioctl(SIOCADDTUNNEL, dev, remote.s_addr, local.s_addr,
                        IPPROTO_IPIP, nas_addr);
}

static int tunnel_ioctl(int cmd, const char *dev, __u32 remote, __u32 local,
                  int proto, __u32 key)
{
.....
      if (proto == IPPROTO_IPIP && key != 0) {
            p.nas_addr = key;
            dynamics_strlcpy(ifr.ifr_name, dev, IFNAMSIZ);
            ifr.ifr_ifru.ifru_data = (void*) &p;
            if (ioctl(s, SIOCSETTUNNEL, &ifr) != 0) {
                  close(s);
                  return -1;
            }
      }
...
}
```

Figure 6 - 31: Dynamics Internal Function I

- 行動連結資訊建立 (Binding Create)

這部分牽涉到兩個檔案，首先當行動端註冊時會呼叫原始碼路徑/dynmacis/src/ha 下之檔案 ha.c 內的 create_binding 函式。 該函式內會呼叫原始碼路徑/dynmacis/src/other 下之檔案 tunnel.c 內負責建立合適的封裝通道模式函式 tunnel_add。 因此，如圖 6-27 所示修改過後的函式，在呼叫 tunnel_add 函式時我們會將行動端的轉交位址放在最後的參數欄位內。 而 tunnel_add 函式本身便檢查該參數存在與否呼叫上述用來建立 IP^{HO} 承載 IP 通道的函式 dyn_ip_tunnel_add_nas 並把該參數，也就是行動端的轉交位址傳遞下去作為型態 252 之標頭選項內所要填入的 IP 位址。

```
static struct bindingentry *
create_binding(struct msg_extensions *ext, struct spi_entry *mn_spi,
            int create_tunnels, struct sockaddr_in cli_addr)
{
.....
     if (ok && create_tunnels){
              if (ext->req->co_addr.s_addr !=  cli_addr.sin_addr.s_addr){
                    if (tunnel_add(tunnels, binding->lower_addr, binding->tun_dev,
                       ext->req->ha_addr, 1, TUNNEL_IPIP, (int)ext->req->co_addr.s_addr) == NULL) {
                    LOG2(LOG_ERR, "Could not add tunnel to FA %s (COA=%s) for "
                    "MN %s\n", inet_ntoa(binding->lower_addr), co_addrstr,
                    mn_addrstr);
                    ok = FALSE;
                    }
              }
.....
 }

TUNNEL *
tunnel_add(HASH *hash, struct in_addr dst_addr, char *dev,
          struct in_addr local_addr, int set_link_up, int type, int key )
{
.....
     if (type == TUNNEL_IPIP) {
              if(key){
                    if (dyn_ip_tunnel_add_nas(device, dst_addr, local_addr,key) != 0) {
                    DEBUG(DEBUG_FLAG, "\tdyn_ip_tunnel_add_nas failed!\n");
                    return NULL;
                    }
              }
.....
}
```

Figure 6 - 32: Dynamics Internal Function II

- Network Address Swapping Function

在 Linux 的系統中，netfilter 是一個用來做封包擷取的架構，它在封包通過協定堆疊的途中定義了許多攔截點(hook). 以 IPv4 來說，封包的處理流程與經過的攔截點如圖 6-29 所示。

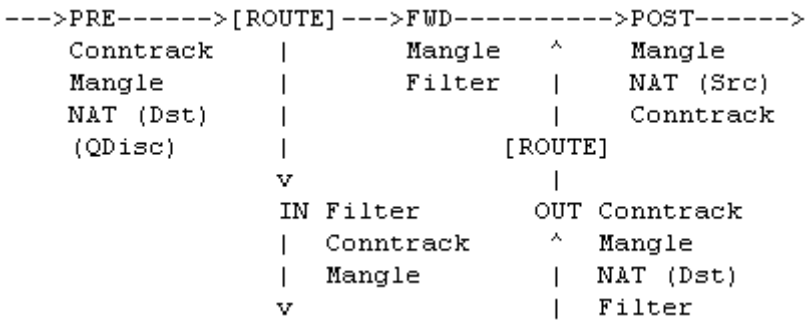

Figure 6 - 33: IPv4 Traversal Diagram

iptables 便是建立在 netfilter 架構之上的封包篩選系統。 核心模組可以註冊新的 table 並要求封包經過該 table。 封包篩選的方法可用作封包的過濾(filter table)與網路位址轉譯(nat table)等。對於後者來說，便是利用攔截點 NF_IP_PRE_ROUTING 與 NF_IP_POST_ROUTING 執行非本機所產生之封包的轉譯目的與來源 IP 位址。 因此，要實作我們所提出的網路位址交換模組便必須修改這兩個攔截點的處理函式。 以下便介紹對此二攔截點的修改：

- 資料結構 (Data Structure)

  為了讓網路位址轉譯器認得我們所定義的 IP 標頭選項，我們同樣必須在系統中定義其資料結構。 因此，在核心原始碼的路徑/usr/src/linux/include/linux/netfilter_ipv4 下，我們加入了如同圖 6-24 之資料結構的檔案 ip_nas.h。

- 攔截點 (Hook Point)

  上述兩攔截點所對應的函式在原始碼路徑/usr/src/linux/net/ipv4/netfilter 下的 ip_nat_core.c 檔案內。 圖 6-30 內的函式 manip_pkt 便是負責執行轉譯 IP 位址的函式，當變數 maniptype 為 IP_NAT_MANIP_SRC 即表示這是處理經過攔截點 NF_IP_POST_ROUTING 的封包，也就是由網路位址轉譯器內部送往外部的封包，反之則為經過攔截點 NF_IP_PRE_ROUTING 的封包。 因此當發現封包帶有型態 252 的 IP 標頭選項時便根據 5.4.2 所訂定的規則執行網路位址交換。 否則，僅執行原本的網路位址轉譯。

```
static void
manip_pkt(u_int16_t proto, struct iphdr *iph, size_t len,
        const struct ip_conntrack_manip *manip,
        enum ip_nat_manip_type maniptype,
        __u32 *nfcache)
{
.....
      if (maniptype == IP_NAT_MANIP_SRC) {
            if (iph->ihl > 5) {
                  struct ip_nat_as_opt *aso = (struct ip_nat_as_opt *)((u_int32_t *)iph + 5);

                  if (aso->code == IPOPT_AS && aso->length >= 8 && aso->addr == INADDR_ANY) {
                        u_int32_t oldval = *(u_int32_t *)aso;

                        iph->check = ip_nat_cheat_check(INADDR_NONE, manip->ip, iph->check);
                        aso->addr = iph->saddr;
                        iph->saddr = manip->ip;
                        aso->count++;
                        iph->check = ip_nat_cheat_check(~oldval, *(u_int32_t *)aso, iph->check);
                  }
            } else {
            iph->check = ip_nat_cheat_check(~iph->saddr, manip->ip,
                                    iph->check);
            iph->saddr = manip->ip;
            }
      } else {
            if (iph->ihl > 5) {
                  struct ip_nat_as_opt *aso = (struct ip_nat_as_opt *)((u_int32_t *)iph + 5);

                  if (aso->code == IPOPT_AS && aso->length >= 8 && aso->addr == manip->ip) {
                        u_int32_t oldval = *(u_int32_t *)aso;

                        aso->addr = iph->daddr;
                        iph->daddr = manip->ip;
                        aso->count++;
                        iph->check = ip_nat_cheat_check(~oldval, *(u_int32_t *)aso, iph->check);
                  }
            } else {
            iph->check = ip_nat_cheat_check(~iph->daddr, manip->ip,
                                    iph->check);
            iph->daddr = manip->ip;
            }
      }
.....
}
```

Figure 6 - 34: NAT Internal Function

6.2.3 Software Demonstration

在本節的最後我們將以實際的程式展示 5.4.3 的 IP^{HO} 承載 IP 通道模式，並用 ICMP 回應請求/回應回覆訊息做為實例。

- Software Configuration
  - 家代理器 (Home Agent)

    在家代理器的部分必須先載入 IP-in-IP^{HO} 的封裝模組後再啟動家代理器的主程式，如圖 6-31 所示。

Figure 6 - 35: Home Agent Configuration Snapshot

  - 行動端 (Mobile Node)

    行動端的部分則是在執行主程式的時候指定不使用無線網路的延伸功能並先啟始在離線的狀態。 接著以用來控制主程式運作的公用程式 dynmn_tool 指定行動端使用配置轉交位址模式向家代理器註冊，如下圖所示。

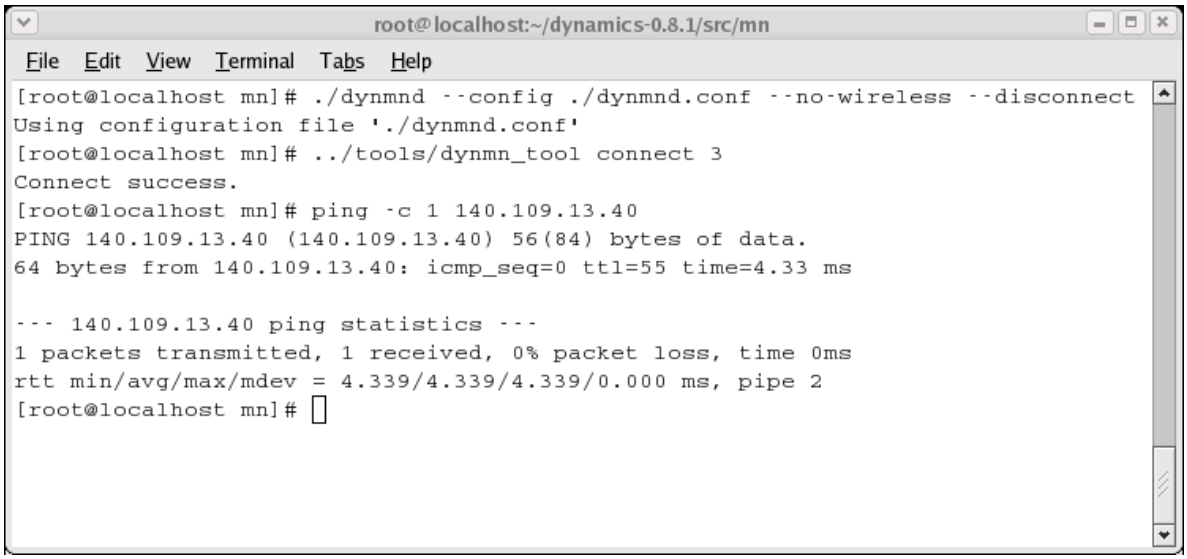

Figure 6 - 36: Mobile Node Configuration Snapshot

■ Program Snapshots

承上，當完成註冊後我們由行動端向通訊端發送一個 ICMP 的回應請求訊息，然後檢視該封包在經過網路位址轉譯器時的處理。 如圖 6-33 所示，左邊的為網路位址轉譯器對外網路卡 eth0 上之封包擷取，而右邊的則為對內網路卡 eth1 上之封包擷取。 誠如 5.4.3 的敘述，該封包首先經過 IP 承載 IP 的封裝後送出於對內的網路卡上，也就是下圖右方的第一個封包。 由該封包的內容我們可以注意到行動端目前的配置轉交位址為 10.113.215.181 (0x0a71d7b5)。 當封包經過位址轉譯後便出現在對外的網路卡上，也就是下圖左方的第一個封包，此時該封包的來源位址已由原本的行動端配置轉交位址換為網路位址轉譯器的公有 IP 位址 140.113.17.15 (0x8c71110f)。 若干時間後，當家代理器收到通訊端的 ICMP 的回應回覆訊息，便以 IP^{HO} 承載 IP 的封裝方式將該封包送出，而該封包便回到網路位址轉譯器的對外網路卡上，也就是下圖左方的第二個封包。 同樣由該封包的內容我們可以注意到在外層的 IP 標頭部分多了型態 252 的標頭選項，而且裡面所放置的 IP 位址為行動端目前的配置轉交位址。 最後，當該封包出現在對內的網路卡上時，也就是下圖右方的第二個封包，型態 252 標頭選項內的 IP 位址已與外層 IP 標頭的目的 IP 位址交換，完全符合我們所預期的結果。 因此，行動端便順利地得到該 ICMP 的回應回覆訊息(見圖 6-32)，同時也驗證了我們所提出方法的可行性。

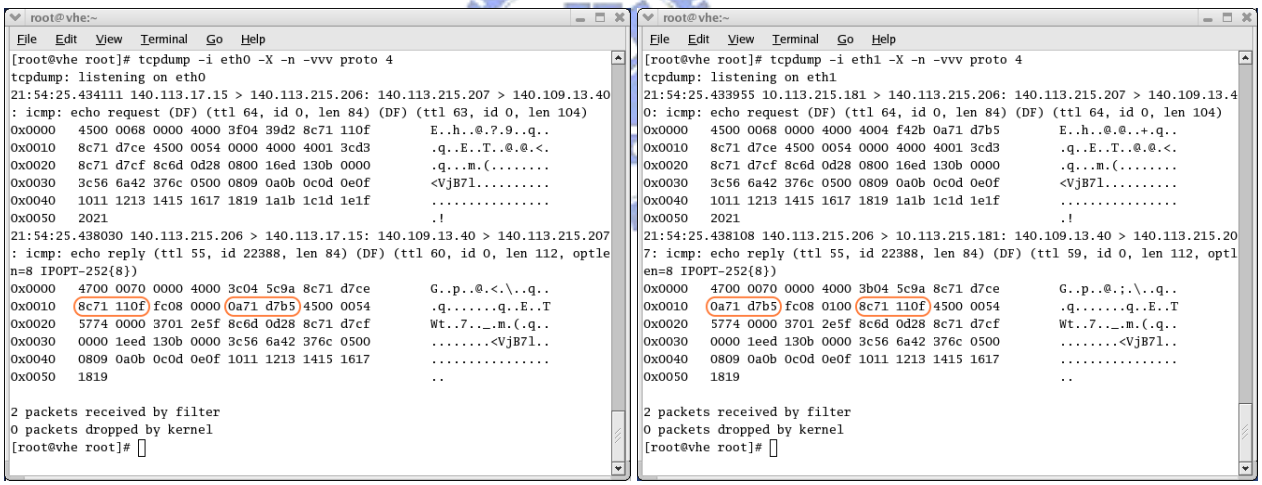

Figure 6 - 37: Packet Dump on External /Internal Interface

Chapter 7 Discussion and Remark

Section 7.1 Multi-Interface Integration Alternatives

在第四章中我們已經詳細的介紹實作整合異質網路平台所需的軟體元件與其所執行的功能。在本節中將就行動端家位址之設定，封包之封裝與鏈結層資訊之取得等議題探討是否有其它的解決方案並比較其與我們所提出方法的優缺點。

7.1.1 Home IP Address Configuration

由於我們是以鬆散連結的整合架構來整合異質網路，因此必須靠行動網際網路協定來支援漫遊時的行動管理。 而根據該協定行動端必須隨時皆使用家位址與其它的通訊端建立連線方可避免切換網路時所造成的連線中斷。 換句話說，行動端的家位址必須隨時地設定於行動端的某張網路卡上供網路應用程式來使用。 我們的作法是以 IP 別名的技巧將家位址加在主配接卡上，但這樣的方法會遭遇到的問題就是媒體感應事件會造成該 IP 位址的遺失。 於是我們便以 Mobile IP 服務中間層驅動程式來克服這樣的問題。

不過，我們也可考慮以 DDK 所提供的範例程式"虛擬迷你連接埠驅動程式 (Virtual Miniport Driver)"來解決該問題。 就如同它的名稱，該驅動程式主要是實作了一張虛擬的網路卡，由於缺乏實體的硬體裝置，原本的媒體感應事件自然不復存在。 所以我們便可利用這樣的特性，將行動端的家位址設定在該網路卡上，這樣一來便不再需要 Mobile IP 服務中間層驅動程式。 然而，使用虛擬迷你連接埠驅動程式卻也產生新的問題： 當行動端回到家網域時，由於家位址設定在該虛擬網路卡上，根據路由的規則所有送往家網域中其它主機的封包接會經由該驅動程式處理。 在缺乏真正網路卡的情況下，封包無法透過該驅動程式送出或接收。 所以我們必須透過 NDIS 中間層驅動程式或 NDIS 攔截驅動程式將送至該虛擬迷你連接埠驅動程式的封包予以擷取並導至其餘的實體網路卡執行實際的封包傳送。 不光是如此，負責執行實際封包傳送的實體網路卡還必須替該虛擬網路卡執行代理(Proxy) ARP 的工作。

由上面的解說不難看出，替代方案雖然可以免除媒體感應所造成的問題，但卻也增加了系統的複雜度。 因為無論在家網域或是外地網域都必須執行封包的擷取動作，相較於我們原有的方法只有在外地網域時才需要攔截封包。 然而，替代方案的好處在於我們可以不受限制地隨時抽換系統所使用的實體網路卡。 原本主配接卡與副配接卡的差異在於前者設定有行動端的家位址，因此在系統運作中不可被移除。 不過當有了虛擬網路卡後，由於其擁有行動端的家位址便是我們所謂的主配接卡，而其餘的實體網路卡自然皆成為副配接卡。 於是，在這狀況下，使用者便可以視所在的環境抽換使用的網路卡，或是在省電的考量下關閉某些網路卡的電源而不影響系統的運作。

7.1.2 Packet Encapsulation

在封包的處理方面，我們使用了 NdisFlt 之 NDIS 攔截驅動程式配合 IpFlt 驅動程式在用戶模式下實作行動網際網路協定所需要的 IP 承載 IP 或是 UDP 承載 IP 之封裝。 另外的作法是將上述的這些驅動程式整合成單一的 NDIS 中間層驅動程式，也就是在核心模式下執行封包的封裝。 它的好處是簡化了封包的處理流程。 在我們原有的方法中封包會由核心模式所擷取後傳回用戶模式處理封裝再由核心模式所送出，如此一來便可能產生多次記憶體暫存空間的複製動作。 而如果是採用替代方案，封包在下到核心模式後便在其內處理封裝與後續的動作，無須再回到用戶模式，因此避免了繁瑣與費時的複製行為。

不過，使用中間層驅動程式並非全然的沒有缺點。 我們知道，在封裝封包外層的 IP 標頭時，必須根據行動端目前的配置轉交位址與家代理器的 IP 位址，且在送出該封裝後的封包之前，還需經過路由與實體位址解譯的過程產生合適的乙太網路標頭。 因此假如是在用戶模式下執行封包的封裝動作，我們便可很自然地取得與運行上述存在於 TCP/IP 協定驅動程式內的相關資訊與動作。然而，如果是採行中間層驅動程式的話，由於它位在協定驅動程式之下，便無法達成如同上述的行為。 換句話說，我們需要透過較複雜的方式，如使用裝置控制命令將封裝時所需要的 IP 標頭與乙太網路標頭資訊告知中間層驅動程式。

7.1.3 Link Layer Information

在我們的實作當中是用 NdisProt 之協定驅動程式來取得鏈結層的資訊，其中包含媒體的訊號強度與連結狀態等。 該資訊實際上是儲存在迷你連接埠驅動程式內，因此只要是位在其上的任何 NDIS 驅動程式皆可以適當的 NDIS 函式來取得之。 換句話說，我們也可用中間層驅動程式來達到相同的功能。 至於採用協定驅動程式或中間層驅動程式，對於其上使用之的用戶模式應用程式來說不管在功能面或是操作面並沒有任何的差別，純粹只是程式設計時的考量。 中間層驅動程式必須同時兼顧協定層驅動程式與迷你連接埠驅動程式，而協定驅動程式只需要注意迷你連接埠驅動程式即可。

Section 7.2 Performance Evaluation of Mobile IP NAT Traversal

本節中將就理論上頻寬的使用率，實際上傳輸的數值與備援能力等議題探討 IETF 所提出的 UDP 承載 IP 通道與我們所提出的 IP^{HO} 承載 IP 通道兩者之間的效能比較。

7.2.1 Bandwidth Utilization

接下來讓我們考慮在頻寬為 100 Mbits/s 的乙太網路上傳送封包的兩種情況：

- 封包封裝後未達最大傳輸單位

當應用程式所送出的原始封包經過封裝後未達最大傳輸單位，意即該封包無須分割便可傳送到網路上。 現假設原始的封包為一普通的 ICMP 回應請求訊息，也就是長度為 78 個位元組，其中 IP 標頭的部分佔 20 個位元組而乙太網路標頭加上檢查碼的部分共佔 18 個位元組。 首先考慮 UDP 承載 IP 的封裝方式，我們必須加入長度為 4 個位元組的行動網際網路協定訊息標頭，長度為 8 個位元組的 UDP 標頭以及最外層長度為 20 個位元組的 IP 標頭。 總計下來一共需要花費 32 個位元組在封裝的額外負擔上，也就是佔整體封包的(32/(78+32))*100% = 29.09%，因此實際有效頻寬便僅剩下 70.91Mbits/s。 但如果是採用 IP^{HO} 承載 IP 的封裝方式，我們只需要加上長度為 28 個位元組的 IP 標頭即可，其中已包含長度為 8 個位元組的標頭選項。 換句話說，在封裝上的額外負擔只佔了(28/(78+28))*100% = 26.42%，所以實際有效頻寬便增加為 73.58Mbits/s。 透過圖 7-1 我們可以更清楚地瞭解兩種不同的封裝方式額外標頭部分所佔用頻寬的情形。

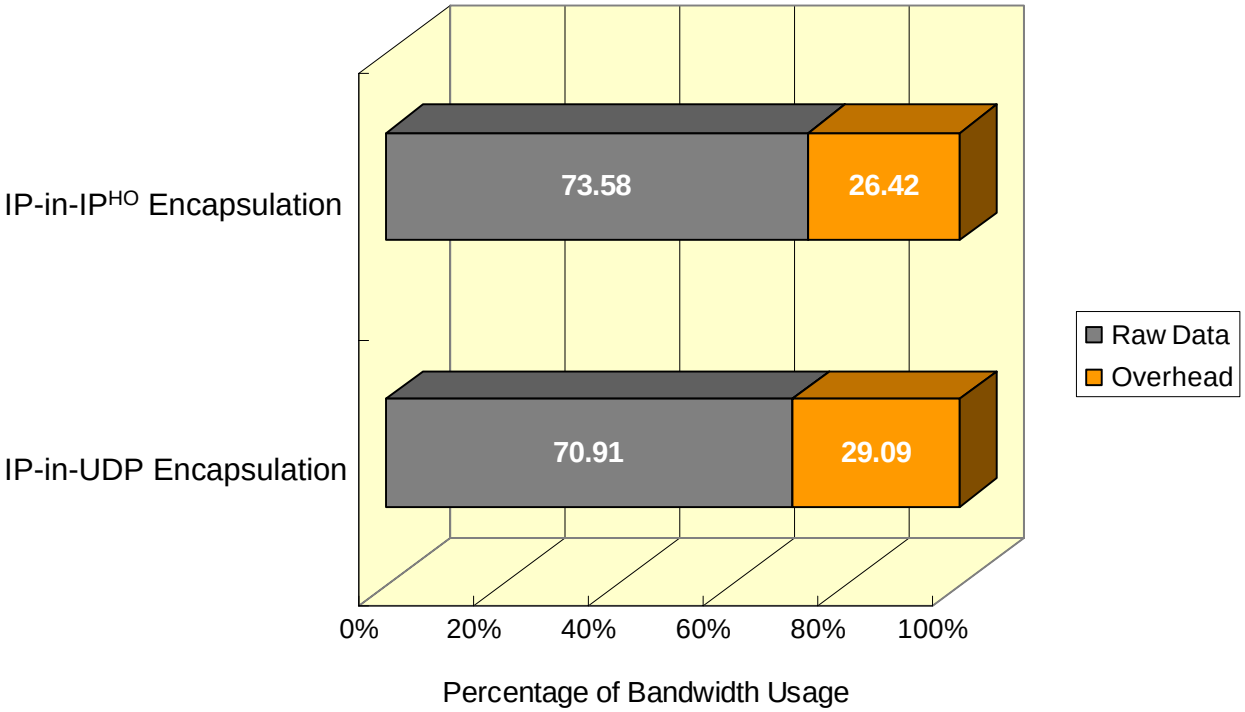

Figure 7 - 1: Bandwidth Utilization Comparison on Packet Length not Exceeding MTU

■ 封包封裝後超過最大傳輸單位

當應用程式所送出的原始封包經過封裝後超過最大傳輸單位，意即該封包必須被分割成兩個皆小於或等於最大傳輸單位的封包後方可分別傳送到網路上。 現假設原始的封包長度為最乙太網路的大傳輸單位，也就是 1518 個位元組，同樣其中 IP 標頭的部分佔 20 個位元組而乙太網路標頭加上檢查碼的部分共佔 18 個位元組。 因此在分割時，第一個封包的長度為 1486 個位元組，內含乙太網路標頭與 IP 標頭，而第二個封包長度為 32 個位元組，其內純粹為數據資料，需要加入額外的乙太網路標頭與 IP 標頭。 首先考慮 UDP 承載 IP 的封裝方式，由於有兩個封包，所以我們一共需要兩個長度為 4 個位元組的行動網際網路協定訊息標頭，兩個長度為 8 個位元組的 UDP 標頭，兩個長度為 20 個位元組。 此外由於第二個封包缺乏 IP 標頭及乙太網路標頭與檢查碼，也必須補上。 總計下來一共需要花費 102 個位元組在封裝的額外負擔上，也就是佔整體封包的 (102/(1518+102))*100% = 6.30%，因此實際有效頻寬便僅剩下 93.70Mbits/s。 但如果是採用 IP^{HO} 承載 IP 的封裝方式，我們只需要額外的兩個包含標頭選項之 IP 標頭，個別的長度皆為 28 個位元組以及為第二個封包準備的標準 20 個位元組之 IP 標頭與長度為 18 個位元組之乙太網路標頭加上檢查碼。 換句話說，在封裝上的額外負擔只佔了(94/(1518+94))*100% = 5.83%，所以實際有效頻寬便增加為 94.17Mbits/s。 透過圖 7-2 我們可以更清楚地瞭解兩種不同的封裝方式額外標頭部分所佔用頻寬的情形。

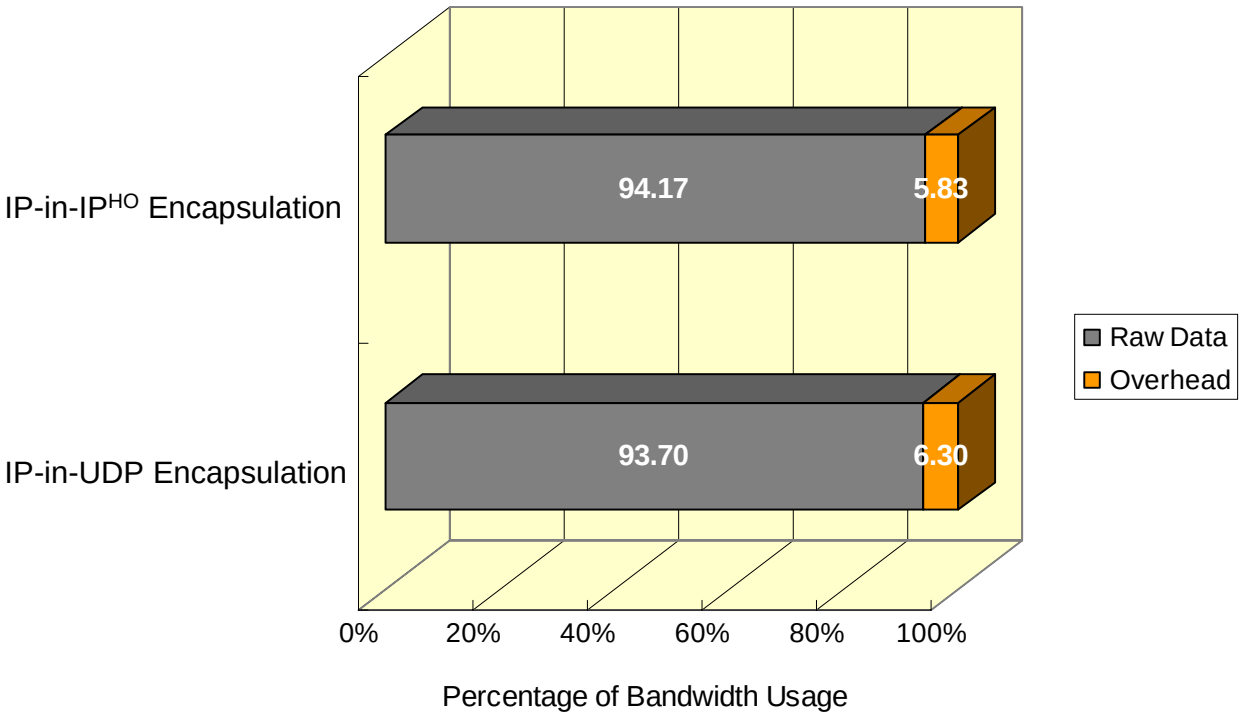

Figure 7 - 2: Bandwidth Utilization Comparison on Packet Length Exceeding MTU

透過以上的分析比較可以知道由我們所提出的 IP^{HO} 承載 IP 之封裝方式無論在何種情況下的有效頻寬使用率皆高於現有的 UDP 承載 IP 封裝方式。

7.2.2 TCP Throughput Estimation

為了比較兩種封裝模式運用在實際傳輸資料時的效能，我們建立如圖 7-3 所示的一個封閉式網路環境。 該環境中只存在家代理器、行動端、通訊端、網路位址轉譯器與 DHCP 伺服器等必要的主機，因此可避免不相關的通訊影響測試過程所得到的數據。 其中家代理器與網路位址轉譯器之間是以 100MB/s 的乙太網路線直接連接，並在雙方路由表上設有彼此的路由。 測試的流程則為先由行動端分別建立 UDP 承載 IP 通道與 IP^{HO} 承載 IP 通道並採用反向通道技術。 接著行動端以 FTP 客戶端程式 lftp 連線通訊端上的 FTP 伺服器 vsftpd 抓取大小為 1.2GB 的檔案並記錄使用不同封裝方式下的平均的傳輸速度。 同時，為了避免硬碟存取速度所造成的誤差，我們並非真的將該檔案儲存在硬碟上而是導入/dev/null。 如此便可純粹地表現出實際的傳輸速度。

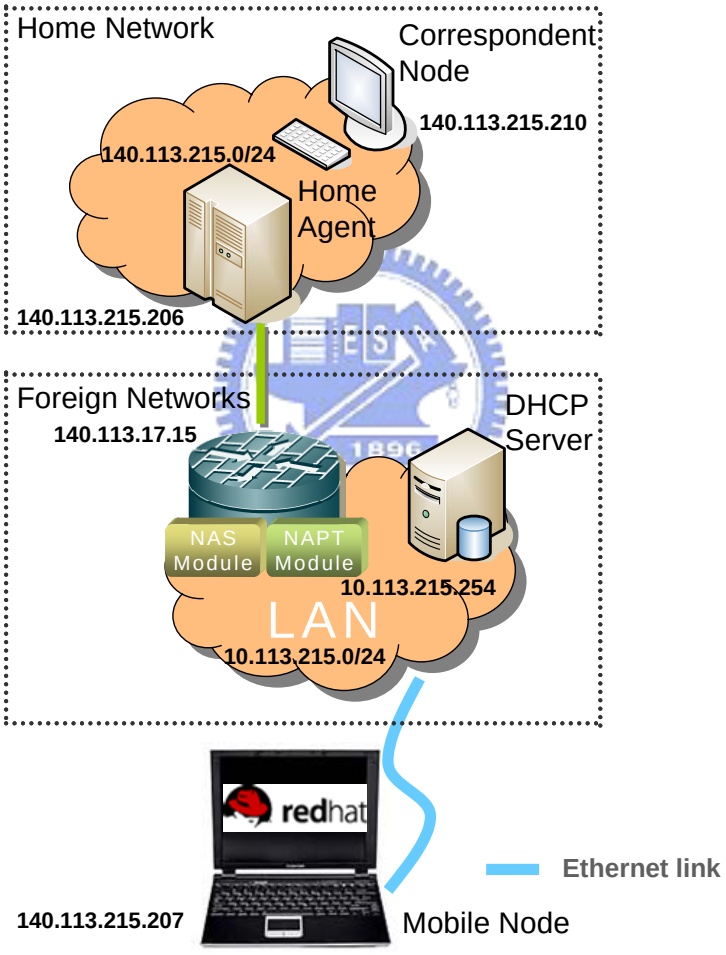

Figure 7 - 3: TCP Throughput Estimation Testbed

圖 7-4 便是傳輸十次所得到的測試結果，由圖中可看出 UDP 承載 IP 的封裝方式所得到的平均傳輸速度為 10.82MB/s，而 IP^{HO} 承載 IP 的封裝方式則為 10.97MB/s。 由此可證實我們所提出的方法確實在傳輸的效能上高出了 1.39%。

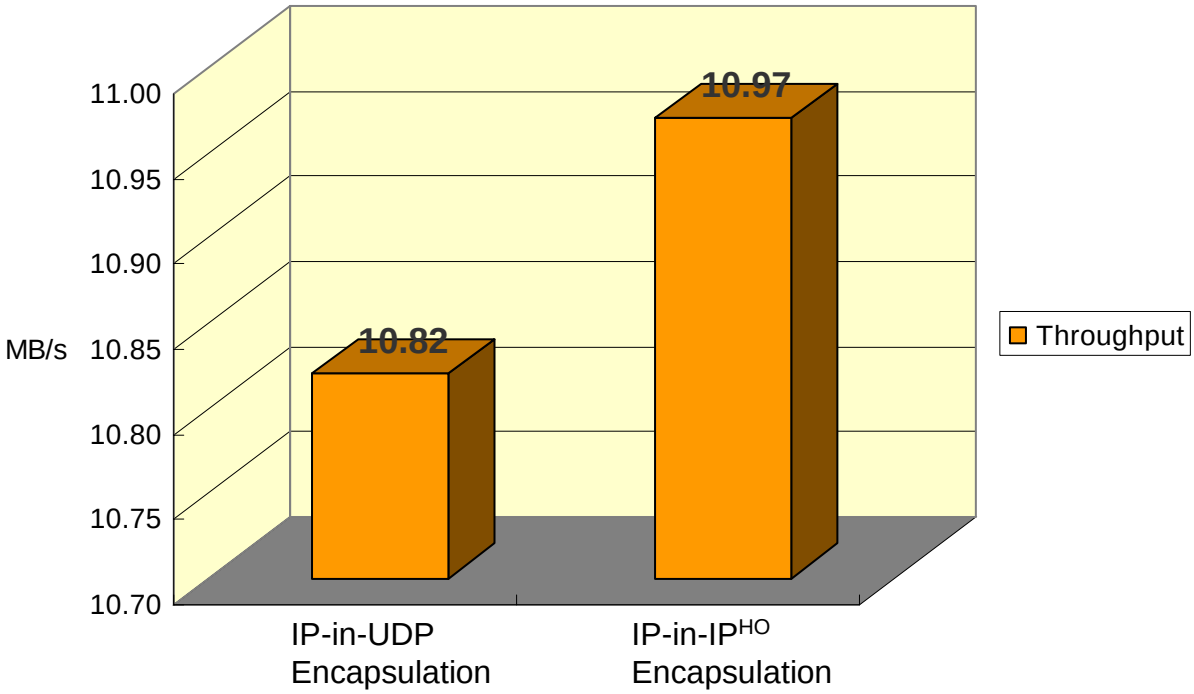

Figure 7 - 4: TCP Throughput Estimation Result

7.2.3 Fault Tolerence Cability

讓我們考慮一個實際的情況，當行動端在位於網路位址轉譯器之下的外地網域向家代理器完成註冊後。 該網路位址轉譯器基於某種原因使得它必須被重新啟動或以另外的機器取代。 此時行動端或是家代理器並沒有任何的機制可發現該行為的發生，於是乎將不會有新的註冊要求訊息被送出。 假如先前是以 UDP 承載 IP 方式所建立的通道，由於網路位址轉譯器上的轉譯表已被清空，原有的網路位址與埠號對應關係便不再存在，因此，經過封裝後的封包到達網路位址轉譯器時便會被丟棄。 直到行動端所註冊的有效時間到期後再次地送出新的註冊要求訊息，原有的網路位址與埠號對應關係才會便重新建立。 換句話說，在這段期間內恐將會造成連線的中斷。 相反的，如果先前是採用我們所提出以 IP^{HO} 承載 IP 方式所建立的通道，由於我們並不仰賴網路位址轉譯器上的網路位址與埠號對應關係。 所以只要其上的網路位址交換模組維持運作，上述的情況將不會對我們造成任何的影響，我們也無須用額外的動作來恢復原本所建立的通道。 也就是說，應用程式如採行 IP^{HO} 承載 IP 的封裝來穿越網路位址轉譯器，無形中該網路位址轉譯器將相對的具備容錯或是備援能力。

透過以上的分析比較我們可以歸納出，我們所提出的方法唯一的缺點是必須修改現有的網路位址轉譯器裝置，但該修改並不影響原有的功能。 且修改過後的網路位址轉譯器不但在效能上有所提升進而也增加了它本身的容錯或是備援能力。

Chapter 8 Conclusion and Future Work

Section 8.1 Conclusion

自從英代爾推出迅馳行動運算技術之後，已經有越來越多的行動終端裝置如筆記型電腦，平板電腦,個人數位助理甚至是智慧型手機皆擁有異質的網路介面卡可供連線至網際網路。 這意味著使用者對於隨時隨地存取網路的需求日益提升。 在本篇論文中便隨著這項趨勢設計了一套異質網路漫遊系統的整合平台，夠過該平台所提供的自動網路偵測與切換機制，使用者不僅可以享受到擁有異質網路介面卡所帶來更寬廣的網際網路存取範圍，同時更可以享有漫遊時的連線不中斷服務。

本平台的客戶端部分是在微軟的作業系統 Windows XP (Service Pack 2)上所開發，採行鬆散連結式的整合架構並以行動網際網路協定作為漫遊時的行動管理機制，因此在運作上還需搭配芬蘭赫爾辛基科技大學所開發的 Dynamics 之行動代理伺服器。 至於所支援的異質網路包含乙太網路(Ethernet)、無線區域網路(WiFi)、整合封包無線電服務(GPRS)、個人手持電話系統(PHS)與第三代全球行動電話系統等。 使用者除了可以透過平台所提供的圖形化操作介面得知目前系統依照底層媒體連結狀態所自動選用的配接卡及其相關的網路狀態資訊外，也可依照自己的喜好指定使用的配接卡。 此外，在應用程式的配合上，現有絕大部分的網路程式皆無須經過任何的修改便可在此平台上運行並享有平台所提供的服務。

除了以上所描述的功能外，平台的另一項特色就是支援私有網域，也就是網路位址轉譯器的使用。 誠如大家的瞭解，隨著網際網路的興起，主機與設備皆呈現高速度的成長導致原有 IPv4 的網路位址已不敷使用。 現今最普遍的解決方案便是採用網路位址轉譯器，雖然它延長了 IPv4 位址的使用壽命，但也破壞了許多既有網路協定的運作。 行動網際網路協定便是其一，因此 IETF 也在隨後提出了因應的 IP-in-UDP 封裝解決方案，本平台即是採用此標準方法來克服此問題。 但是我們認為穿越網路位址轉譯器仍有更佳的解決方式，於是在本論文中也提出了另外的 IP-in-IP^{HO} 封裝來證實同樣也可克服行動網際網路協定運作於網路位址轉譯器內並可得到比原有解決方案更好的效能。

Section 8.2 Future Work

在開發本異質網路漫遊系統的整合平台之時，我們預留了極大的空間讓程式設計師可以實作各式各樣的切換抉擇演算法。 我們在文中已展示了一種以優先權為考量的方法，然而更進一步我們可以加入媒體特性的考量，例如以訊號強度作為切換判斷的準則。 除此之外，我們還可以配合具位置知覺的快速切換(Location-awared Fast Handoff)演算法改善在無線區域網路間切換所造成的延遲。 或是為每個造訪的網路建立偏好設定檔(Profile)，讓系統根據該設定檔在連線到該網路時套用相關的設定，例如自動啟用虛擬私有網路(Virtual Private Network, VPN)連線等。 此外，基於安全性的考量，未來也可在系統的運作上加上認證授權與計費的機制。

另一方面，目前可攜式裝置上的作業系統除了筆記型電腦外，大多是微軟專為該類型裝置所開發的 Windows CE .NET。 該作業系統可說是 Windows XP 的精簡版本，因此無論是在整體的系統架構或是網路元件的部分皆有些許的差異。 未來如果能成功移植本平台至該作業系統，將可大大地展現其運用的空間，並可激發更多運行於其上的應用程式，諸如網際網路電話，網路視迅或及時訊息等。 讓使用者真正感受到異質網路整合所帶來的好處與更多可享受的服務。

最後在我們所設計的以 IP 標頭選項穿越網路位址轉譯器的方法上，由於我們所強調的是相較於原有的網路位址埠號轉譯模組，網路位址交換模組可省略在封包經過網路位址轉譯器時的查表動作。 因此理論上，當轉譯表的內容越大，也就是越多的連線被建立時，將會更凸顯我們的方法在效能上的改進。 然而，在現實的考量下卻很難建立這樣的測試環境，因為需要太多的設備來達成。 因此，未來可以考慮將我們的實作重新實現在 ns2 的模擬器上，藉由其便可設置出我們所需要的虛擬實驗環境。 在該環境下重新地檢視 UDP 承載 IP 與 IP^{HO} 承載 IP 兩封裝方式在網路位址轉譯器滿載的狀況下之表現，如此將可以更加地佐證我們所提出方法的優異點所在。

Appendix A Reference

[1] J. Postel, “User Datagram Protocol,” IETF RFC 768, August 1980.
[2] J. Postel, “Internet Protocol,” IETF RFC 791, September 1981.
[3] S. Deering, Ed., “ICMP Router Discovery Messages,” IETF RFC 1256, September 1991.
[4] K. Egevang, P. Francis, “The IP Network Address Translator (NAT),” IETF RFC 1631, May 1994.
[5] J. Reynolds, J. Postel, “Assigned Numbers,” IETF RFC 1700, October 1994.
[6] S. Hanks, T. Li, D. Farinacci, P. Traina, “Generic Routing Encapsulation (GRE),” IETF RFC 1701, October 1994.
[7] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, E. Lear, “Address Allocation for Private Internets,” IETF RFC 1918, February 1996.
[8] C. Perkins, Ed., “IP Mobility Support,” IETF RFC 2002, October 1996.
[9] C. Perkins, Ed., “IP Encapsulation within IP,” IETF RFC 2003, October 1996.
[10] C. Perkins, Ed., “Minimal Encapsulation within IP,” IETF RFC 2004, October 1996.
[11] H. Krawczyk, M. Bellare, R. Canetti, “HMAC: Keyed-Hashing for Message Authentication,” IETF RFC 2104, February 1997.
[12] R. Droms, “Dynamic Host Configuration Protocol,” IETF RFC 2131, March 1997.
[13] G. Montenegro, Ed., “Reverse Tunneling for Mobile IP, revised,” IETF RFC 3024, January 2001.
[14] T. Hiller, P. Walsh, X. Chen, M. Munson, G. Dommety, S. Sivalingham, B. Lim, P. McCann, H. Shiino, B. Hirschman, S. Manning, R. Hsu, H. Koo, M. Lipford, P. Calhoun, C. Lo, E. Jaques, E. Campbell, Y. Xu, S. Baba, T. Ayaki, T. Seki, A. Hameed, “CDMA2000 Wireless Data Requirements for AAA,” IETF RFC 3141, June 2001.
[15] C. Perkins, Ed., “IP Mobility Support for IPv4,” IETF RFC 3220, January 2002.
[16] C. Perkins, Ed., “IP Mobility Support for IPv4,” IETF RFC 3344, August 2002.
[17] H. Levkowetz, S. Vaarala, “Mobile IP Traversal of Network Address Translation (NAT) Devices,” IETF RFC 3519, April 2003.
[18] Karim El Malki, Ed., Pat R. Calhoun, Tom Hiller, James Kempf, Peter J. McCann, Ajoy Singh, Hesham Soliman, Sebastian Thalanany, “Low Latency Handoffs in Mobile IPv4,” IETF Internet Draft, June 2002. draft-ietf-mobileip-lowlatency-handoffs-v4-04.txt.
[19] “Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications,” ANSI/IEEE Std. 802.11 Part 11, ISO/IEC 8802-11:1999(E), 1999.
[20] 3GPP, “General Packet Radio Service (GPRS) Service Description, Stage 1,” TS22.060 V 3.5.0, October 2000.
[21] 3GPP, “General Packet Radio Service (GPRS) Service Description, Stage 2,” TS23.060 V 5.1.0, March 2002.
[22] 3GPP2, “Removable User Identity Module (R-UIM) for CDMA 2000 Spread Spectrum Systems,” C.S0023-0, IS-820, June 2000.
[23] 3GPP2, “CDMA 2000 Wireless IP Network Standard,” P.S0001-B, TIA-835-B-1[E], October 2000.
[24] ETSI, “Broadband Radio Access Networks (BRAN); HIPERLAN Type 2; Requirements and Architectures for Interworking between Hiperlan/2 and 3rd Generation Cellular Systems,” TR 101 957 V1.1.1, August 2001.
[25] M. Buddhikot, G. Chandranmenon, S. J. Han, Y. W. Lee, S. Miller and L. Salgarelli, “Integration of 802.11 and Third Generation Wirelesss Data Networks,” Proceedings of IEEE Infocom 2003, April 2003.
[26] M. Buddhikot, G. Chandranmenon, S. J. Han, Y. W. Lee, S. Miller and L. Salgarelli, “Design and Implementation of a WLAN/CDMA2000 Integration Architecture,” Special Issue of IEEE Communications Magazine on 3G+802.11 Integration, November 2003.
[27] M. Adda, J. Saunders, A. Peart, “A Simple Implementation to Roaming between GPRS and WiFi Networks,” IADAT International Conference on Telecommunications and Computer Networks, December 2004.

[28] M. Adda, J. Saunders, “Seamless Roaming between GPRS and WiFi Networks,” The 1st Thailand Computer Science Conference, December 2004.
[29] T.C. van Sebille, “WLAN - GPRS Roaming, Based on Mobile IPv4,” Vodafone “Integrating WLAN Solutions into a Macro Cellular Network” Project Public Report, October 2002.
[30] OSR Online, “Kernel: Calling User Mode - Using the Inverted Call Model,” The NT Insider, Vol 9, Issue 1, February 2002.
[31] Microsoft Windows Driver Development Kit (DDK), http://msdn.microsoft.com/library/en-us/ddkint/hh/ddkint/ddksplash_0d0ef7c0-7411-4fed-8c52-ef4690fe6e40.xml.asp, 1985.
[32] The Network Simulator ns-2, http://www.isi.edu/nsnam/ns/, 1995.
[33] Dynamics Mobile IP 0.8.1, http://dynamics.sourceforge.net, October 2001.
[34] Simple NDIS Hooking Based Firewall for NT4/2000, http://ntdev.h1.ru/ndis_fw.html, June 2003.
[35] IpHook, http://www.hollistech.com/Resources/IpHook/hts_iphook.htm, January 2000.