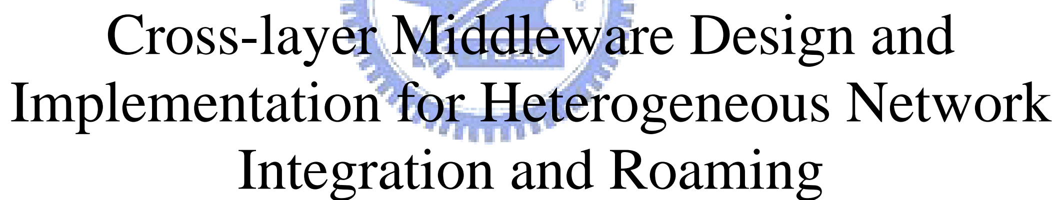
# 國立交通大學

## 資訊工程學系

## 碩 士 論 文

# 異質網路整合與漫遊之跨階層中介軟體設計與實作



# Cross-layer Middleware Design and Implementation for Heterogeneous Network Integration and Roaming


研 究 生：李俊儀

指導教授：曾建超 教授



中華民國九十四年六月

# 異質網路整合與漫遊之跨階層中介軟體設計與實作

# Cross-layer Middleware Design and Implementation for Heterogeneous Network Integration and Roaming

研 究 生：李俊儀　　　　Student：Chun-I Lee

指導教授：曾建超　　　　Advisor：Chien-Chao Tseng

國 立 交 通 大 學

資 訊 工 程 系

碩 士 論 文

A Thesis

Submitted to Department of Computer Science and Information Engeering

College of Electrical Engineering and Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer and Information Science

June 2005

Hsinchu, Taiwan, Republic of China

中華民國九十四年六月

# 異質網路整合與漫遊之跨階層中介軟體設計與實作

研究生：李俊儀　　　　　　　　　　　　　　指導教授：曾建超

國立交通大學資訊工程學系碩士班

## 摘　　　要

隨著許多無線接取技術和服務的成熟，我們已經可以隨時隨地透過無線網路存網際網路。同時，在行動裝置上配備多種異質無線網路介面以成為趨勢。例如市面上許多智慧型行動電話和個人數位助理已經裝置了無線區域網路介面和 GPRS 介面。但是這些行動裝置卻往往缺乏一個有效的行動管理機制，來同時充分利用這些異質網路介面。為了解決這個問題，我們在以 Windows CE 為基礎的手持裝置上，利用 Mobile IP 網路協定，設計和實作了一套能夠有效整合異質網路，能在異質網路間漫遊和做無接縫的換手的中介軟體。

另一方面，在設計行動管理機制時通常會遇到一個嚴重的問題，就是行動管理程式缺乏底層網路的資訊，而不能做有效的行動管理，例如造成在異質網路間換手時會產生嚴重延遲，以至於不能提供良好的服務品質。而為了增加行動管理的效率，減少換手時的延遲，我們的中介軟體導入了跨階層網路設計的概念。我們的中介軟體能提供在應用層的行動管理程式底層網路的資訊，並透過我們設計的事件觸發機制，讓在應用層的行動管理程式能夠快速的被告知底層網路的改變，並做出適當的反應，而能提供良好的服務品質。

# Cross-layer Middleware Design and Implementation for Heterogeneous Network Integration and Roaming

Student: Chun-I Lee Advisor: Dr. Chien-Chao Tseng

Department of Computer Science and Information Engineering
National Chiao Tung University

ABSTRACT

Because of recent advances in wireless access technologies and services, we can access Internet anytime and anywhere. Furthermore, it is now a trend to equip a mobile device with multiple network interfaces of different wireless accessing technologies. For instance, there are many off the shelve Smartphones and PDAs (personal Digital Assistant) equipped with WLAN (Wireless LAN) and GPRS interfaces. However, in these mobile devices, there is absence of an effective mobility management mechanism to utilize multiple wireless network interfaces simultaneously. In the thesis, we proposed and implemented a middleware to integrate the multiple heterogeneous network interfaces for the handheld device based on Windows CE operating system, so that handheld device can roam and handover between heterogeneous networks seamlessly.

On the other hand, there is often a seriously problem in mobility management. Mobility management is not efficient enough due to lack of underlying network interface status. For example, handover between heterogeneous networks produces long handover delay, so that quality of service can not be guaranteed well. Therefore, we proposed the concept of cross-layer network design in our middleware. Our middleware can provide underlying network information to mobility management program in application layer. In advance, by means of event trigger mechanism we proposed, mobility

management program can be notified underlying network status change rapidly and react to the change promptly, so that quality of service can be guaranteed.

# 誌謝

本論文得以順利完成首要感謝我的指導教授—曾建超博士，在過去兩年對於我耐心的指導。還要感謝實驗室學長姐同學學弟們不吝嗇的幫忙與協助。而在論文寫作過程中，也感謝許多朋友的鼓勵與陪伴。最後僅將此成果獻給我最敬愛且最關心我的母親。

# 目錄

**中文摘要 ........ I**

**英文摘要 ........ II**

**誌謝 ........ IV**

**目錄 ........ V**

**圖目錄 ........ VIII**

**表目錄 ........ X**

**第一章 序論 ........ 1**

1.1 動機 ........ 1

1.2 研究提案和目標 ........ 2

1.3 論文內容 ........ 2

**第二章 背景介紹 ........ 4**

2.1. MOBILE IP ........ 4

*2.1.1 發現代理器 ........ 5*

*2.1.2 註冊 ........ 5*

*2.1.3 通道技術 ........ 5*

2.2 NDIS ........ 9

2.2.1 NDIS 簡介 ........ 9

*2.2.2 迷你連接埠驅動程式 ........ 10*

*2.2.3 通訊協定驅動程式 ........ 12*

*2.2.4 中介層驅動程式 ........ 13*

**第三章 相關論文研究 ........ 17**

3.1 無線異質網路的整合 ........ 17

*3.1.1 異質網路漫遊系統整合平台之設計與實作*......17

*3.1.2 Design and Implementation of a WLAN/cdma2000 Interworking Architecture* ......17

3.2 跨階層網路設計......19

*3. 2. 1 Cross-layer Design in 4G Wireless Terminal*......19

**第四章 移植RIOMIP ......22**

4.1 *RIOMIP* 軟體架構簡介......22

4.2 移植課題......25

*4.2.1 應用程式和裝置驅動程式之間的輸入/輸出(I/O)方式*......25

*4.2.2 封包攔截方式*......30

*4.2.3 應用程式和裝置驅動程式之間的輸入/輸出介面*......31

*4.2.4 動態主機設定協定客戶端函式庫（DHCP Client API）*......33

4.3 替代方案的設計與實作......33

*4.3.1 非同步輸入輸出*......33

*4.3.2 網路中介層驅動程式是實作*......43

*4.3.3 串流介面驅動程式實作*......54

**第五章 RIOMIP之改進 ......61**

5.1 動機......61

5.2 問題分析......61

5.3 解決方案......63

5.4 實作細節......65

*5.4.1. 多路傳輸中介層驅動程式實作*......65

*6.4.2 封包的操作和封裝*......66

**第六章 跨階層網路事件傳遞中介軟體設計與實作 ......68**

6.1 跨階層網路事件傳遞中介軟體概觀......68

6.2 驅動程式和應用程式間訊系的傳遞......69

*6.2.1 共享事件*......69

*6.2.2 轉化呼叫模型 (Inverted Call Model)*......70

*6.2.3 命名事件 (Named Event )* ........ *70*

6.3 跨階層網路事件傳遞中介軟體系統架構........ 71

*6.3.1 L2L3 動態函式庫* ........ *72*

*6.3.2 CLM 協定驅動程式* ........ *73*

6.4 跨階層網路事件傳遞中介軟體運作原理........ 73

*6.4.1 中介軟體實初始化動作* ........ *73*

*6.4.2 查詢網路裝置* ........ *74*

*6.4.3 訂閱網路事件* ........ *74*

*6.4.4 設定和查詢網路狀態變數* ........ *76*

6.5 CLM協定驅動程式的實作 ........ 79

*6.5.1 協定驅動程式的實作* ........ *79*

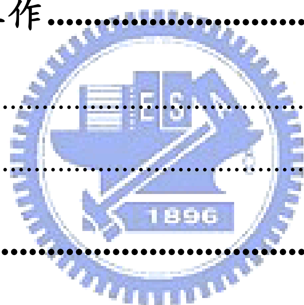
*6.5.2 串流驅動程式的實作* ........ *80*

**第七章 結論和未來工作 ........ 81**

7.1 結論........ 81

7.2 未來工作........ 81

**參考文獻 ........ 82**



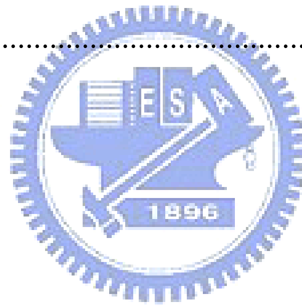
# 圖目錄

圖 2-1 IP承載IP封裝 ..... 6
圖 2-2 最小化IP承載封裝 ..... 7
圖 2-3 通用路由封裝 ..... 8
圖 2-4 外地代理器轉交位址模式通道示意圖 ..... 8
圖 2-5 地代理器轉交位址模式通道示意圖 ..... 9
圖 2-6 NDIS驅動程式層級示意圖 ..... 10
圖 2-7 迷你連接埠驅動程式收送封包互動示意圖 ..... 12
圖 2-8 過濾型中介層驅動程式 ..... 15
圖 2-9 一對多過濾型中介層驅動程式 ..... 16
圖 3-1 IODATA 閘道器架構 ..... 18
圖 3-2 IODATA客戶端架構 ..... 19
圖 3-3 CROSS-LAYER COORDINATION PLANES ..... 20
圖 3-4 跨階層合作模式 ..... 21
圖 4-1 *RIOMIP*軟體架構圖 ..... 23
圖 4-2 SYNCHRONOUS I/O VS. ASYNCHRONOUS I/O ..... 26
圖 4-3 應用程式與驅動程式I/O呼叫流程圖 ..... 27
圖 4-4 *RIOMIP*封包傳送流程圖 ..... 28
圖 4-5 封包傳送程式碼流程對應 ..... 29
圖 4-6 NDIS HOOKING 架構示意圖 ..... 31
圖 4-7 DISPATCH ROUTINE註冊 ..... 32
圖 4-8 WINDOWS CE 虛擬計憶體分配圖 ..... 34
圖 4-9 事件運作模式 ..... 37
圖 4-10 共享事件運作模式 ..... 38
圖 4-11 IOCONTEXT 資料結構修改示意圖 ..... 39
圖 4-12 封包傳遞流程示意圖 ..... 40
圖 4-13 緩衝區資料結構 ..... 41

圖 4-14 封包傳送程式碼流程示意圖 ........................................................................42
圖 4-15 NDIS封包傳送非同步運作模式........................................................................45
圖 4-16 NDIS PACKET資料結構 ........................................................................47
圖 4-17 SINGLE-BUFFER NDIS_PACKET 示意圖........................................................................48
圖 4-18 MULTI-BUFFER NDIS_PACKET示意圖 ........................................................................48
圖 4-19 取的封包內容範例 ........................................................................51
圖 4-20 過濾封包和重新包裝封包的流程範例 ........................................................................53
圖 5-1 RIOMIP外送封包流程 ........................................................................62
圖 5-2 理想外送封包流程 ........................................................................63
圖 5-3 新的多路傳輸中介驅動程式及外送封包流程 ........................................................................64
圖 6-1 跨階層網路事件傳遞中介軟體系統架構........................................................................72
圖 6-2 跨階層網路事件傳遞中介軟體系統架構........................................................................76
圖 6-3 設定和查詢狀態變數運作的流程 ........................................................................79

# 表目錄

表 4-1 IRP輸入輸出的功能型態 ....................................................................................32
表 4-2 NDIS中介層驅動程式介面函式....................................................................................43
表 4-3 NDIS存取封包相關函式....................................................................................49
表 4-4 FILE I/O 函式 ....................................................................................54
表 4-5 STREAM INTERFACE FUNCTIONS ....................................................................................55
表 6-1 跨階層網路事件傳遞中介軟體事件和狀態變數 ....................................................................................69
表 6-2 驅動程式和應用程式間訊系的傳遞方式比較 ....................................................................................71
表 6-3 L2L3 動態函式庫函式表 ....................................................................................73
表 6-4 一般網路型態所共用的OIDS ....................................................................................77
表 6-5 OIDS和事件對應 ....................................................................................78



# 第一章 序論

## 1.1 動機

隨著無線通訊技術的成熟與快速普及，以往夢想的隨時隨地 (Anytime, Anywhere) 連上網際網路已經慢慢得被實現。基於 IEEE802.11 所發展出來的無線區域網路 (Wireless LAN) 不管在企業，學校，家庭，甚至在公眾場合都已經被普遍的使用。攜帶著有無線區域網路裝置的筆記型電腦或手持裝置，我們在許多地方都可以藉由這樣的無線擷取技術連結上網際網路。另一方面，行動電話業者也提供了數據服務，被看作成一個無線廣域網路。透過行動電話，我們也可以在任何時間任何地點連上網際網路。而隨著第三代行動電話系統的開台，我們更可以透過行動電話網路享有比以往行動電話系統更高頻寬的數據服務。

然而，這兩種無線通訊技術和服務也各有其優缺點。無線區域網路被視為短程高頻寬，移動性較差的無線通訊技術；而透過行動電話網路所提供的無線廣域網路服務，則視有著較大的服務範圍和較窄的頻寬，同時也提供較好的移動性。所以如果能整合這兩種無線通訊服務，我們就可以同時得到這兩種無線通訊服。

而在市場上，也越來越多行動裝置同時具備了這兩種的無線擷取技術。因為這兩種無線網路服務都是基於 IP 網路架構，也讓整合這兩種無線網路服務變的可能。而要進一步整合，我們就必需面對如何在這兩種網路間漫遊和換手的問題。因為當我們切換不同的無線網路介面的同時，我們也通常會造成 IP 位址的改變，而這通常造成連線的中斷。而從漫遊的觀點來看，IP 位址的改變也產生如何找到行動使用者的問題。

Mobile IP，一個在 IP 層(網路層)提供行動支援的網路協定，就自然成為一個非常合適的解決方法。透過 Mobile IP 的技術，行動端可以一直使用固定的 IP 位址，無論他在那個網域或者是使用那個無線通訊介面，如此一來便可以解決換手和漫遊的問題。由於 IP 沒有改變，所以當行動端在不同的無線網路介面間切換時，所有的 IP 連線都不會中斷，也不會影響上層應用程式的運作。

另一方面，Mobile IP 並不是唯一的解決方案。像是 SIP(Session Initiation Protocol ) 能在應用層提供行動支援，而也有其他的通訊協定能在傳輸層或網路層解決行動性的問

題。然而這些協定都遇到了相同的問題，就是缺乏底層網路的資訊，而造成在換手時的嚴重延遲，以至於不能提供良好的服務品質。為了解決這些問題，我們也需要一套良好的機制，來讓這些行動管理的通訊協定能夠充分的掌握底層網路的資訊，如此才能透過跨階層網路設計的概念，提供良好的服務品質。

## 1.2 研究提案和目標

本論文的研究目標作主要為下列兩點：

1. 利用 Mobile IP 在手持行動裝置支援異質網路的漫遊和換手。
2. 設計和實作一套跨階層訊息傳遞機制，提供其他行動管理的通訊協定能夠充分掌握底層網路的資訊，充分利用跨階層網路設計的概念，提供良好的行動管理。

因為在將來，手持行動裝置整合多無線網路技術已經是個趨勢，而手持裝置也比筆記型電腦更行動應用的可能。所以我將在以 Windows CE 作業系統為基礎的手持裝置上，實作利用 Mobile IP 支援異質網路的漫遊和換手的系統。Window CE 作業系統為現今手持行動裝置非常普遍的作業系統，它延伸出許多系列的作業系統在不同的應用上。例如數位個人助理 (PDA) 上的 Pocket PC 系列系統和智慧型電話上的 SmartPhone 系列系統，都是以 Windows CE 核心為基礎的作業系統。

另一方面，為了提供其他上層的應用程式或者其他行動管理協定能夠掌握底層的網路資訊，進行跨階層網路的設計，我將在 Windows 系列系統上設計一套
跨階層訊息傳遞機制。我將提供一套中介軟體，讓其他程式開發者藉由我的應用程式開發介面，獲得底層網路的事件和資訊。

藉由上面的研究，希望不僅能夠達到隨時隨地地連接網際網路，更能利用做最佳的無線網路技術，做最佳的連結 (Best Connect)。

## 1.3 論文內容

第一章 序論

描述論文動機和這篇論文期望達到的目標。

第二章 背景介紹

介紹研究過程所用到的背景知識。我將簡介 Mobile IP 協定和在 Windows 系列作業

系統下，實作網路相關模組會用到的 NDIS (Network Driver Interface Specification) 介面函式庫。

第三章 相關論文研究

介紹在相同的問題上其他論文的解決方法。

第四章 移植 *RIOMIP*

介紹移植 *RIOMIP* 系統到 Windows CE 作業平台的過程。

第五章 改進 *RIOMIP*

提出改進原本 *RIOMIP* 的想法和整個實作的內容。

第六章 跨階層網路事件傳遞中介軟體設計與實作

描述設計和實作跨階層網路事件傳遞中介軟體的概念和過程。

第七章 結論和未來工作

對本研究做出結論和提出未來可以進行的方向。

# 第二章 背景介紹

## 2.1. Mobile IP

Mobile IP 是由 C. Perkins 等人於 1996 年在 IETF(Internet Engineering Task Force) 所提出的一套網路協定,讓 IP 資料可以正確的繞送到於網際網路上行動的主機 。在 Mobile IP 裡，在網路上行動的主機，我們稱為行動端點 (Mobile Node)。行動端點可以漫遊於不同網域，改變其網際網路的接點，卻使用固定的 IP 位址。這個固定的 IP 位址被稱做家用位址 (Home Address)，被用以作行動端點的網路識別。另外在 Mobile IP 協定中，額外定義了兩種網路實體：

- 家用代理器 (Home Agent)

  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。

- 外地代理器 (Foreign Agent)

  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(De-tunnel) 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。

而當行動端在外地網域時，會再當地取得另外一個動態的 IP 位址，這個 IP 位址會隨著行動端點移動到不同的網域是而有所不同。這個位址被稱作轉交位址(Care-of-Address，COA)。相對於家用位址用來做為移動端點的識別，轉交位址反映了

行動端點現在的網路位置。

Mobile IP 主要可以分為三個程序，發現代理器 (Agent Discovery)，註冊 (Registration) 和通道技術 (Tunneling)。底下我將針對這三個程序作說明。

### 2.1.1 發現代理器

行動代理器 (包括家用代理器和外地代理器) 會定期在它們所屬的網路當中廣播所謂的代理器公告 (Agent Advertisement)。藉由這樣的機制，來讓行動端點發現它們的存在。行動端點也可以藉此知道它現在所屬的位置，是在家用網域還是外地網域。代理器公告是經由 ICMP 的封包來傳送，而且是利用廣播的方式來發送。

另外行動端點也可以主動發出代理器請求(Agent Solicitation) 來要求當地的行動代理器對行動端點發出代理器公告。代理器請求也是經由 ICMP 的封包來傳送，同時是利用廣播的方式來發送，但是行動代理器所回覆的代理器公告則會回送到發送要求行動端。

### 2.1.2 註冊

當行動端點在外地網路時，他必須告知家用代理器它現在所屬的位置，也就是通知家用代理器行動端點現在的轉交位址。轉交位置的獲得有兩種方式：一種是透過外地代理器的代理器公告所獲得，被稱為外地代理器轉交位置(FA Care-of-Address)；另一種則是行動端自己利用其他的機制或協定 (如 DHCP) 所取得，被稱作共同配置轉交位址 (Co-located Care-of-Address)。而這個告知家用代理器行動端點現在的轉交位址的動作，就被稱為註冊。這個註冊的訊息，是利用 UDP 的封包來傳送，使用 434 公定的連接埠。

### 2.1.3 通道技術

當行動端點在外地網路時，接收資料必須藉由家用代理器的協助。所有傳送給移動端點的封包，都會送到家用網域，經由家用代理器代收，在由家用代理器轉送給行動端點。這個轉送的機制，就被稱作通道技術，因為就好像建立一個通道在家用代理器和行

動動點(或者是外地代理器)之間。

而通道技術是藉由資料封裝（Packet Encapsulation）的技巧來達成。Mobile IP 支援了三種資料封裝的技術：

- IP 承載 IP 封裝（IP within IP Encapsulation）

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 2-1 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。

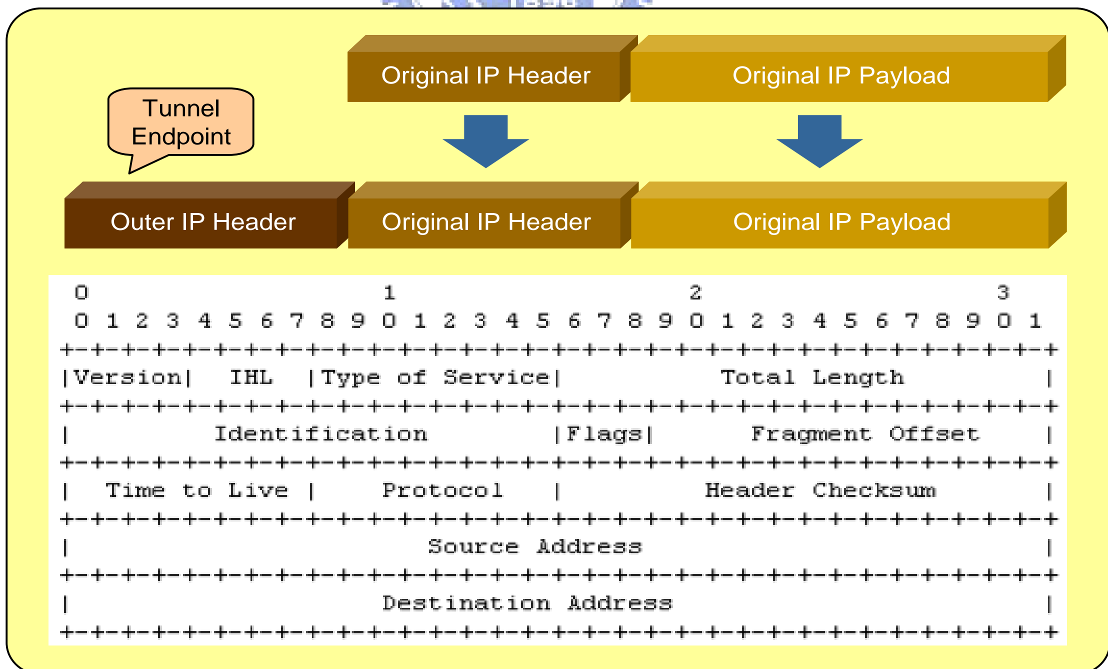



圖 2-1 IP 承載 IP 封裝

- 最小化 IP 承載封裝（Minimal Encapsulation within IP）

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 2-2。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。

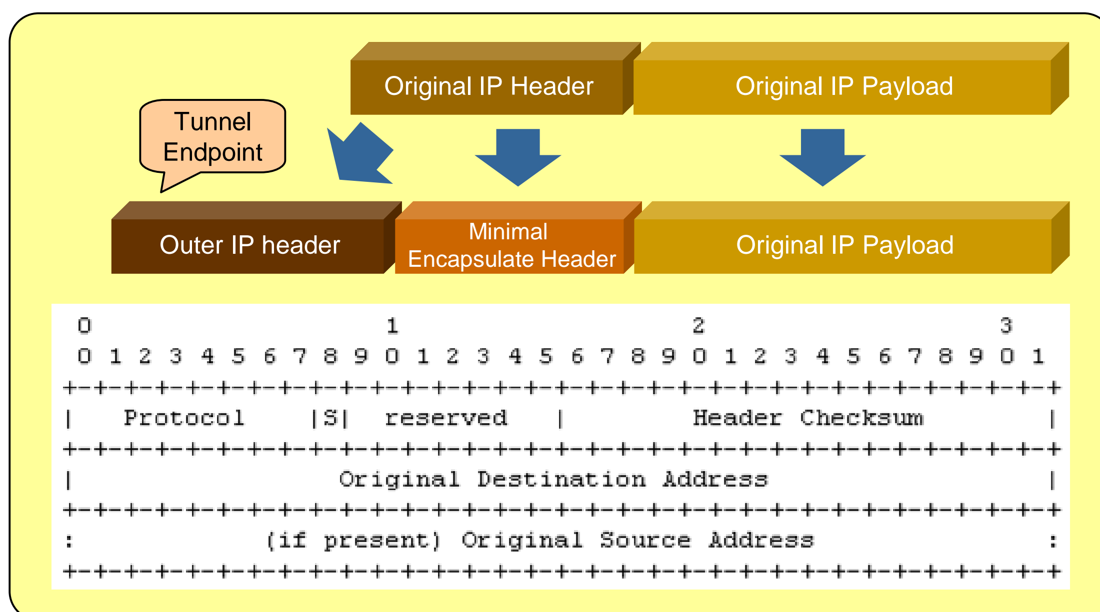



圖 2-2 最小化 IP 承載封裝

- 通用路由封裝（Generic Routing Encapsulation， GRE）

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 2-3。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。

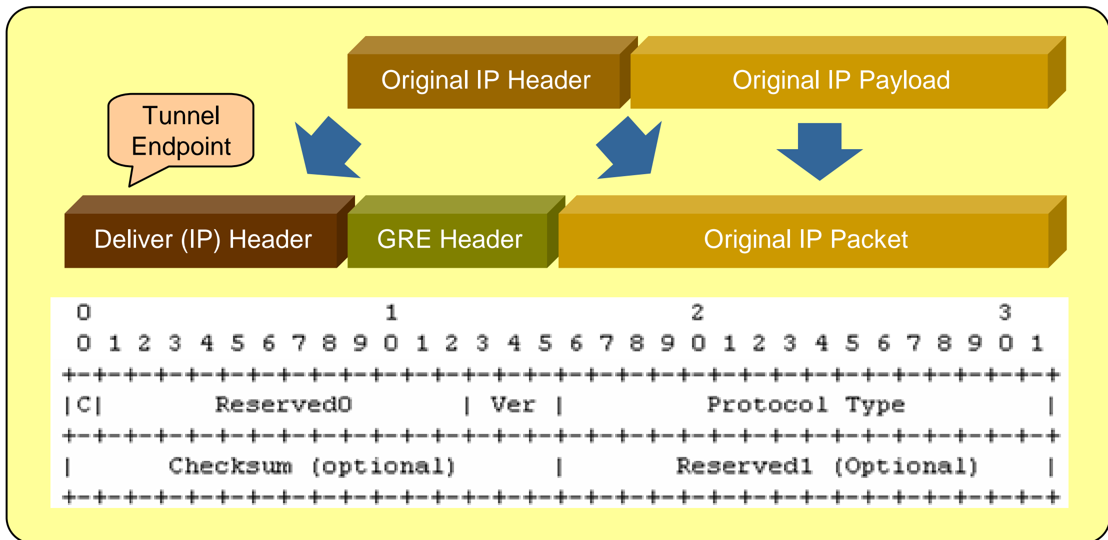



圖 2-3 通用路由封裝

根據取得轉交位址的不同，通道建立的端點也會不同，因此通道的建立被分成兩種模式：

- 外地代理器轉交位址模式

  通道的建立的端點為家用代理器和外地代理器，運作模式如圖 2-4 所示。而在這個模式由外地代理器做封包解封裝的工作。

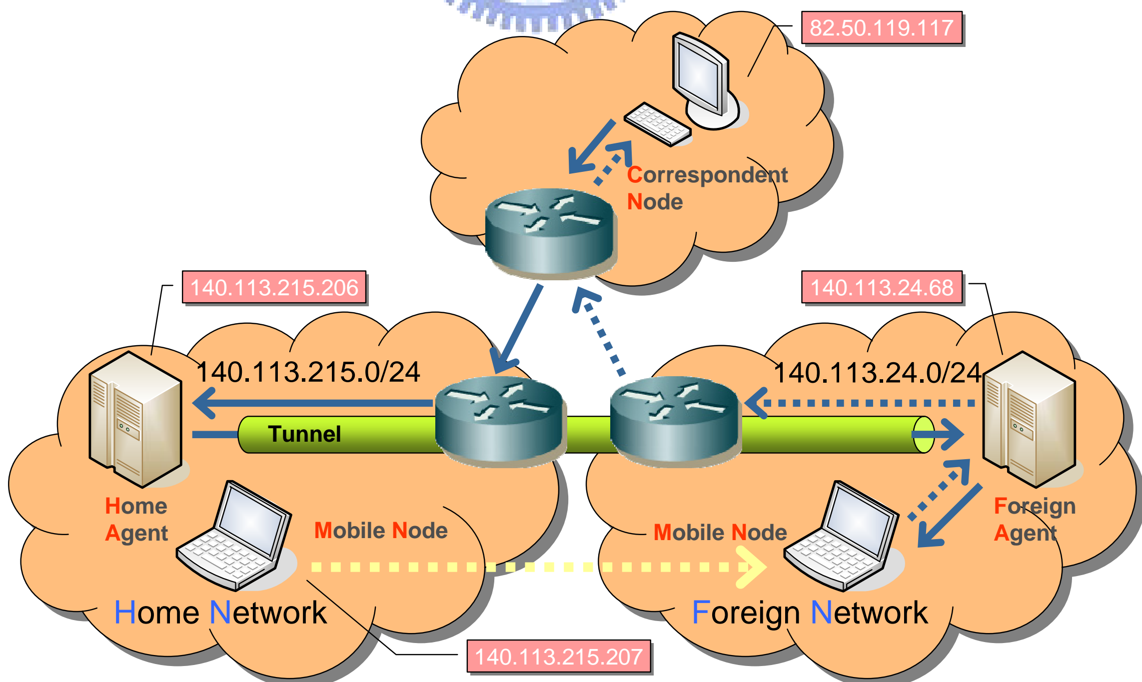



圖 2-4 外地代理器轉交位址模式通道示意圖

- 共同配置轉交位址模式

通道的建立的端點為家用代理器和行動端點本身運作模式如圖 2-x 所示。在這個模式由行動端本身做封包解封裝的工作。

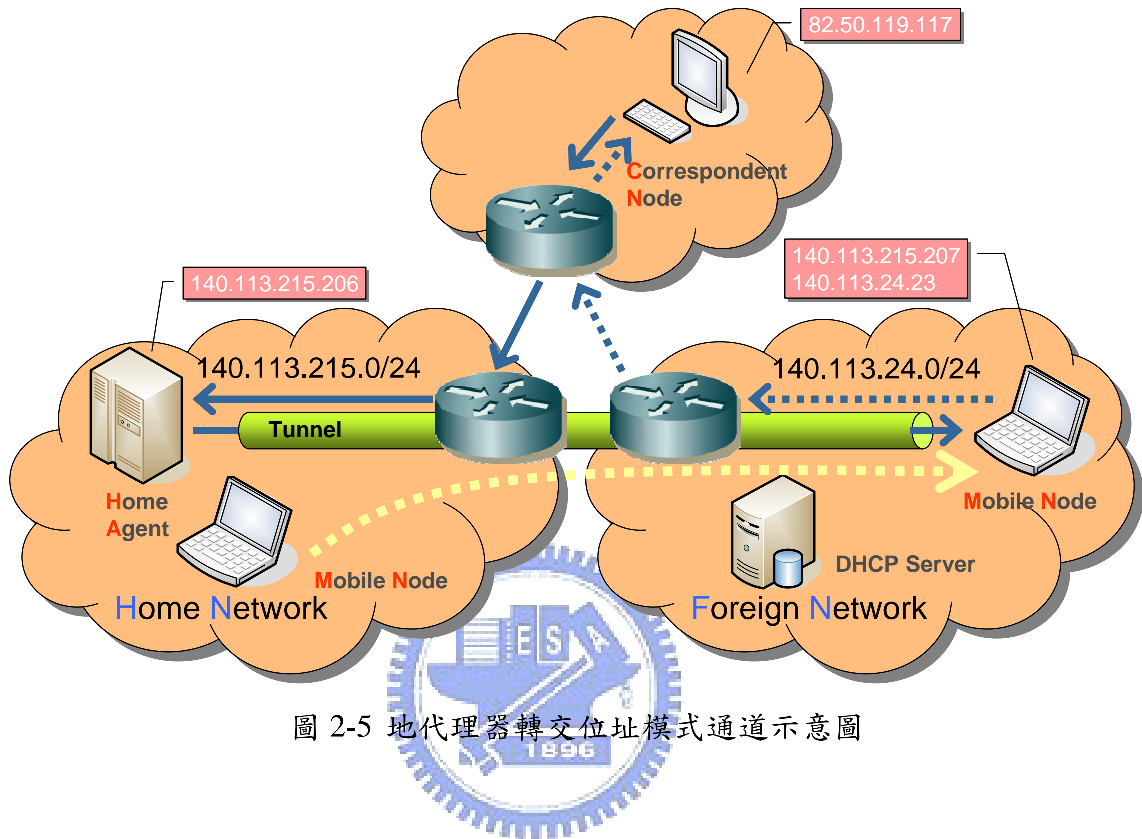



圖 2-5 地代理器轉交位址模式通道示意圖

## 2.2 NDIS

### 2.2.1 NDIS 簡介

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，它抽象化了網路硬體和驅動程式間的介面，同時也抽象化了底層的驅動程式和上層的傳輸協定。它使不同的傳輸協定可以採用一種通用的介面來使用由不同廠商所製造的網路卡。

NDIS 採層級式的架構，它定義了三種不同型態的驅動程式：

- 迷你連接埠驅動程式(Miniport drivers)
- 中介層驅動程式(Intermediate drivers)
- 協定驅動程式(Protocol drivers)

其三種驅動程式的層級概念如下圖 2-6 所示。

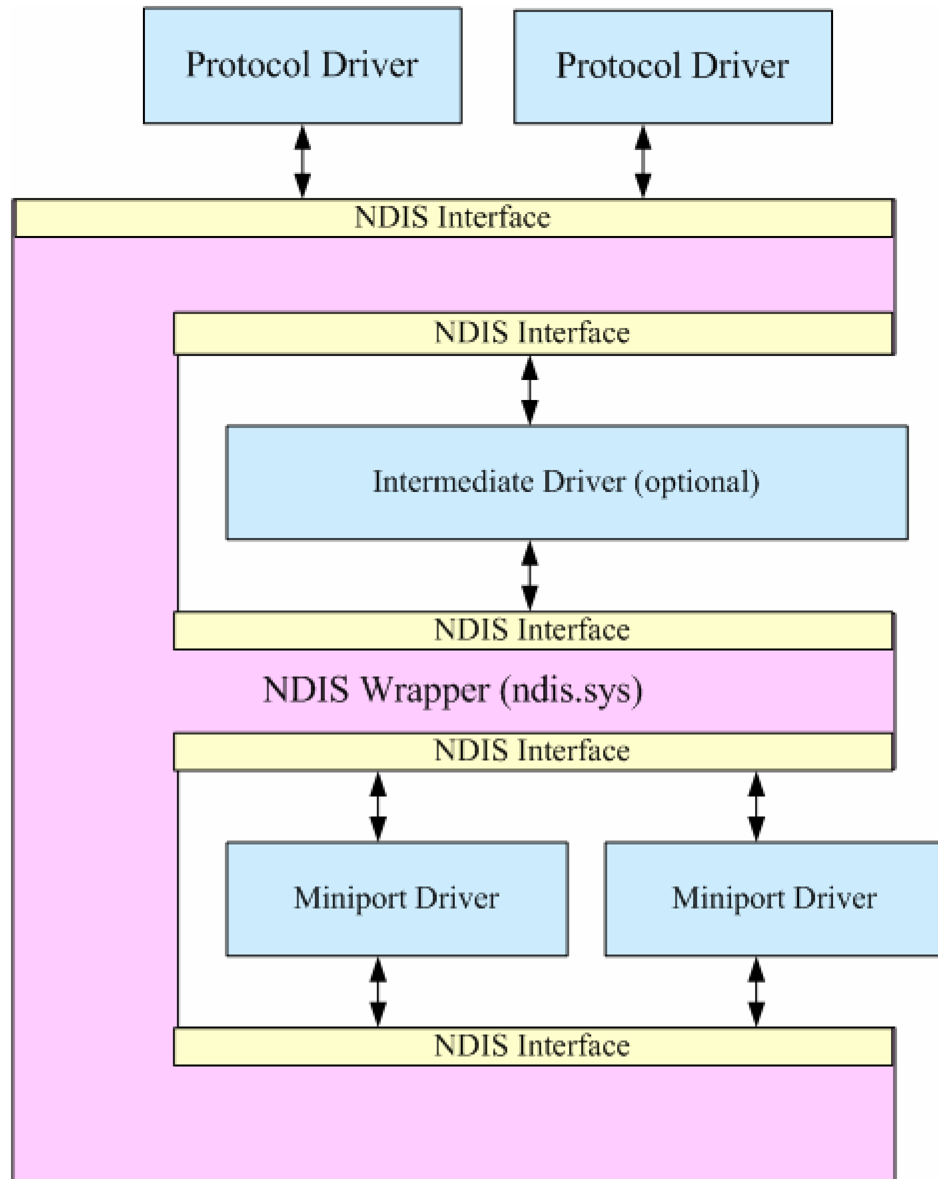



圖 2-6 NDIS 驅動程式層級示意圖

### 2.2.2 迷你連接埠驅動程式

迷你連接埠驅動程式位於 NDIS 網路層級的最下層，有兩個基本的功能：

1. 管理實體網路卡， 包括透過網路卡傳送與接收資料。

2. 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。

迷你連接埠驅動程式都是透過 NDIS 函式庫來和網路介面卡和上層的驅動程式溝通。NDIS 函式庫提供一套完整的函式群(NdisXxx 函式)來包裝所有迷你連接埠所會用到的作業系統所提供的函式。另一方面，迷你連接埠驅動程式必須開放出一套進入點(MiniportXxx 函式) 來讓 NDIS 呼叫，讓上層的驅動程式能夠存取該迷你連接埠驅動程式。

以封包的傳送和接收為例，如圖 2-7 所示，我們可以觀察出迷你連接埠驅動程式是如何和 NDIS 函式庫與上層的驅動程式間做互動：

- 當上層傳輸層的驅動程式要傳送一個封包時，它會呼叫 NDIS 函式庫所開放的函式 NdisXxx。然後 NDIS 函式庫會透過呼叫迷你連接埠驅動程式所開放的適當函式 MiniportXxx，把這個封包傳遞給迷你連接埠驅動程式。最後，迷你連接埠驅動程式在呼叫適當的 NdisXxx 函式把封包透過網路介面卡轉送出去。
- 當底層的網路介面卡接收到封包時，它會產生一個硬體的中斷，這個中斷會被 NDIS 函式庫或者網路介面卡的迷你連接埠驅動程式本身所處理。NDIS 透過呼叫適當的 MiniportXxx 函式來告知該網路介面卡的迷你連接埠驅動程式。迷你連接埠驅動程式從網路介面卡建立要傳送 的資料，迷你連接埠然後通知上層有連結的驅動程式接收封包，透過呼叫 NdisXxx 函式。

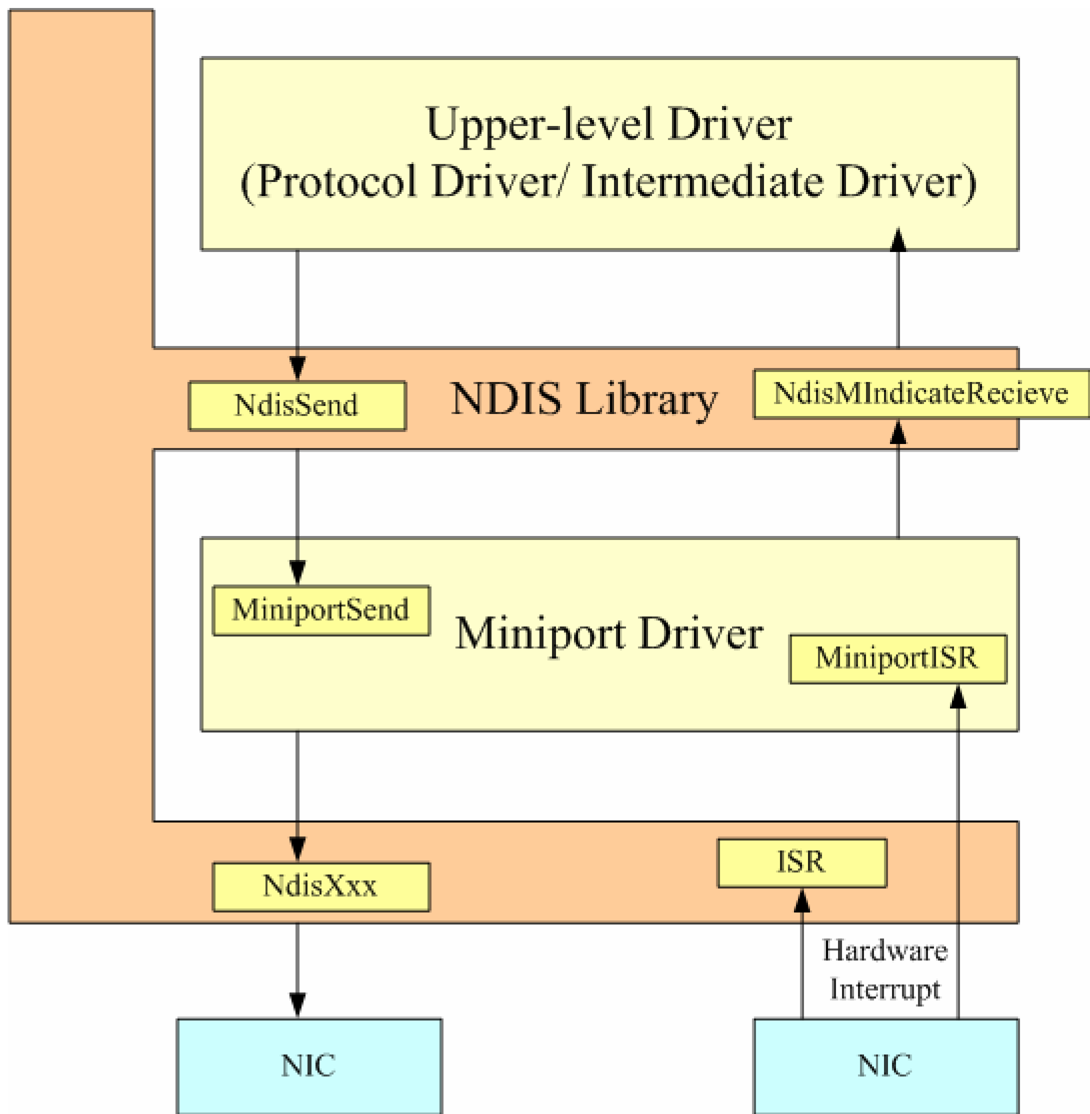



圖 2-7 迷你連接埠驅動程式收送封包互動示意圖

### 2.2.3 通訊協定驅動程式

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。

和迷你連接埠驅動程式類似，協定驅動程式也必須提供介面和底層的驅動程式，可能是迷你連接埠驅動程式或中介曾驅動程式互動。協定驅動程式呼叫 NdisXxx 函式來傳

送封包，讀取和設定底層驅動程式所維護的資訊和使用作業系統的服務。同時，協定驅動程式也必須開放出一套進入點 (ProtocolXxx 函式) ，讓 NDIS 函式庫來呼叫，或讓底層的驅動程式來通知協定驅動程式接收封包及回報底層驅動程式的狀態資訊。

## 2.2.4 中介層驅動程式

中介層驅動程式介於上層的協定驅動程式和下層的迷你連接埠驅動程式之間。通常被用來作下面幾種用途：

- 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

常見的封包排程器就是其中一個例子。封包排程器從每個從上層的協定驅動程式所送下來的封包和從底層迷你連接埠驅動程式所送上來的封包中讀取優先權的資訊，然後作排程的動作，再按照排程的順序往下面的迷你連接埠驅動程式和往上面的通訊協定驅動程式傳送封包。

- 封包傳送負載平衡與容錯備援（Load Balance Fail Over, LBFO）

中介層驅動程式會製造出一張虛擬的介面卡給在其他上面的傳輸驅動程式使用，但是底層透過一張以上的實體介面卡發送封包。

- 監控和收集網路資訊

從由上層協定驅動程式往下送和底層迷你連接埠驅動程式往上送的封包中，擷取資訊，作為監控和統計之用。

由於中介層驅動程式需要同時和上層的協定驅動程式和底層的迷你連接埠驅動程式溝通，所以它必須開放出兩套進入點：

- 協定驅動程式進入點

在中介層驅動程式的底端，NDIS 函式庫會呼叫 ProtoclXxx 函式來和底層的迷你連接埠溝通。對於底層的迷你連接埠驅動程式來說，中介層驅動程式看起來就像協定驅動程式一樣。

- 迷你連接埠驅動程式進入點

在中介層驅動程式的頂端，NDIS 函式庫會呼叫 MiniportXxx 函式來和上層的協定驅動程式溝通。對於上層的協定驅動程式來說，中介層驅動程式看起來就像迷你連接埠驅動程式一樣。

網路中介層驅動程式有兩種型態：過濾中介層驅動程式（NDIS Filter Intermediate Drivers）和多路傳輸中介層驅動程式（MUX intermediate drivers）。底下將分別介紹這兩種型態的中介層驅動程式。

- 過濾中介層驅動程式

過濾中介層驅動程式針對每一個底層的連接埠驅動程式都會產生一個虛擬連接埠。當虛擬連接埠接收請求和封包資料，中介層驅動程式會轉送它們到底曾相關的實體連接埠驅動程式。過濾中介層驅動程式在傳送封包之前或收到封包之後 ，可以去修改封包的內容。例如，過濾中介層驅動程式可以加密和壓縮外送封包的資料， 同時解密和解壓縮接收封包的資料。

底下圖 2-8 解說了過濾型中介層驅動程式的虛擬連接埠裝置和底層實體裝置的迷你連接埠驅動程式一對一連結的關係。

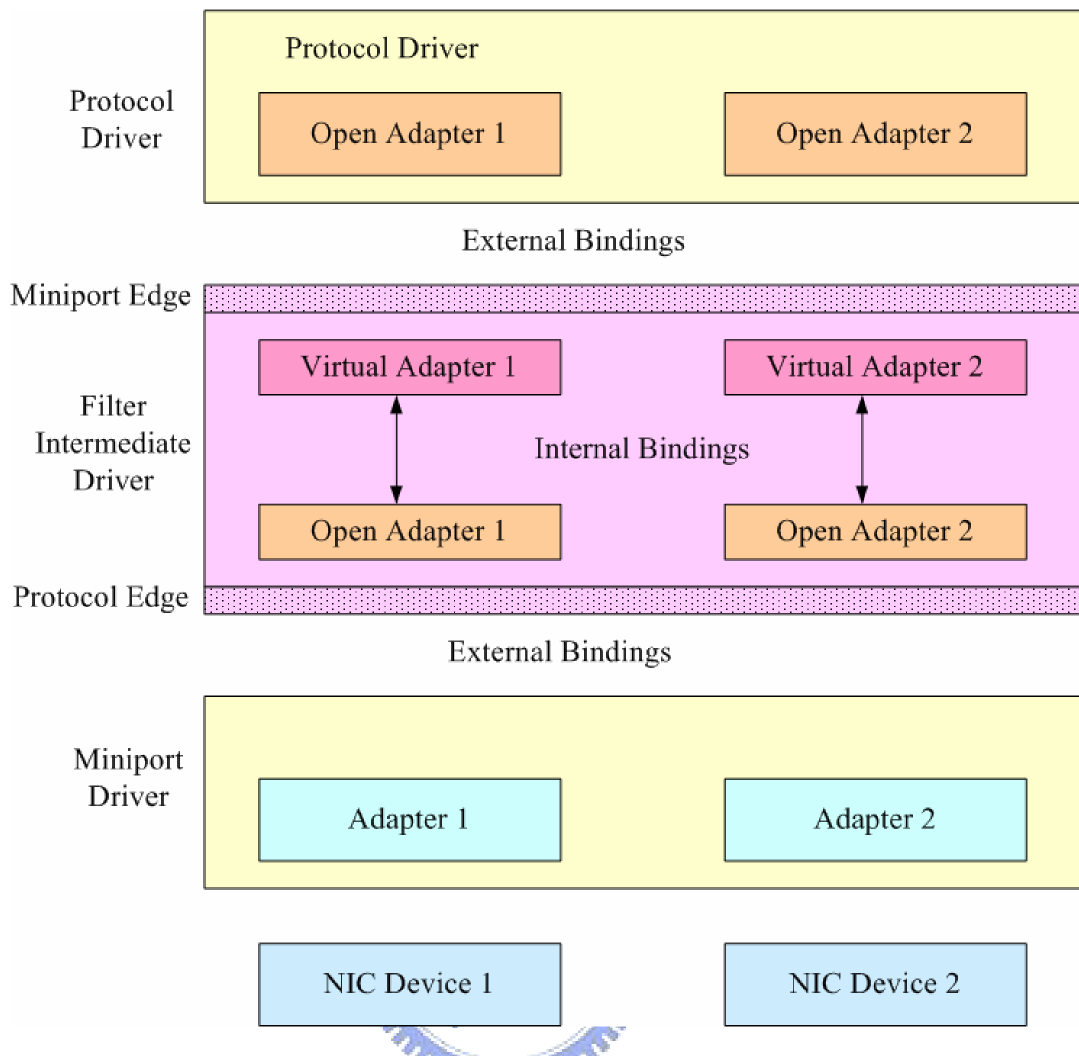



圖 2-8 過濾型中介層驅動程式

- 多路傳輸中介層驅動程式

和過濾中介層驅動程式不一樣的地方， 多路傳輸中介層驅動程式所開放出來的虛擬連接埠驅動程式可以和實際和底層驅動程式連接的網路介面卡數量不一樣。MUX 中介層驅動程式可以用一對多，多對一或者是多對多等和底層介面卡的對應關係來開放虛擬連接埠動程式。這會使的內部連結和資料傳送路徑更為複雜。

以一個一對多的設定為例，一個單一的中介層驅動程式可以連結底層多個實體的網路界面裝置。傳輸層的協定驅動程式連結由中介層驅動程式所產生的虛擬迷你連接埠驅動程式就像連結一般的迷你連接埠驅動程式。多路傳輸中介層驅動程式可以重新包裝和往下傳遞所有的請求和要傳送的封包到某一個實體的迷你連接埠驅

動程式。如圖 2-9。

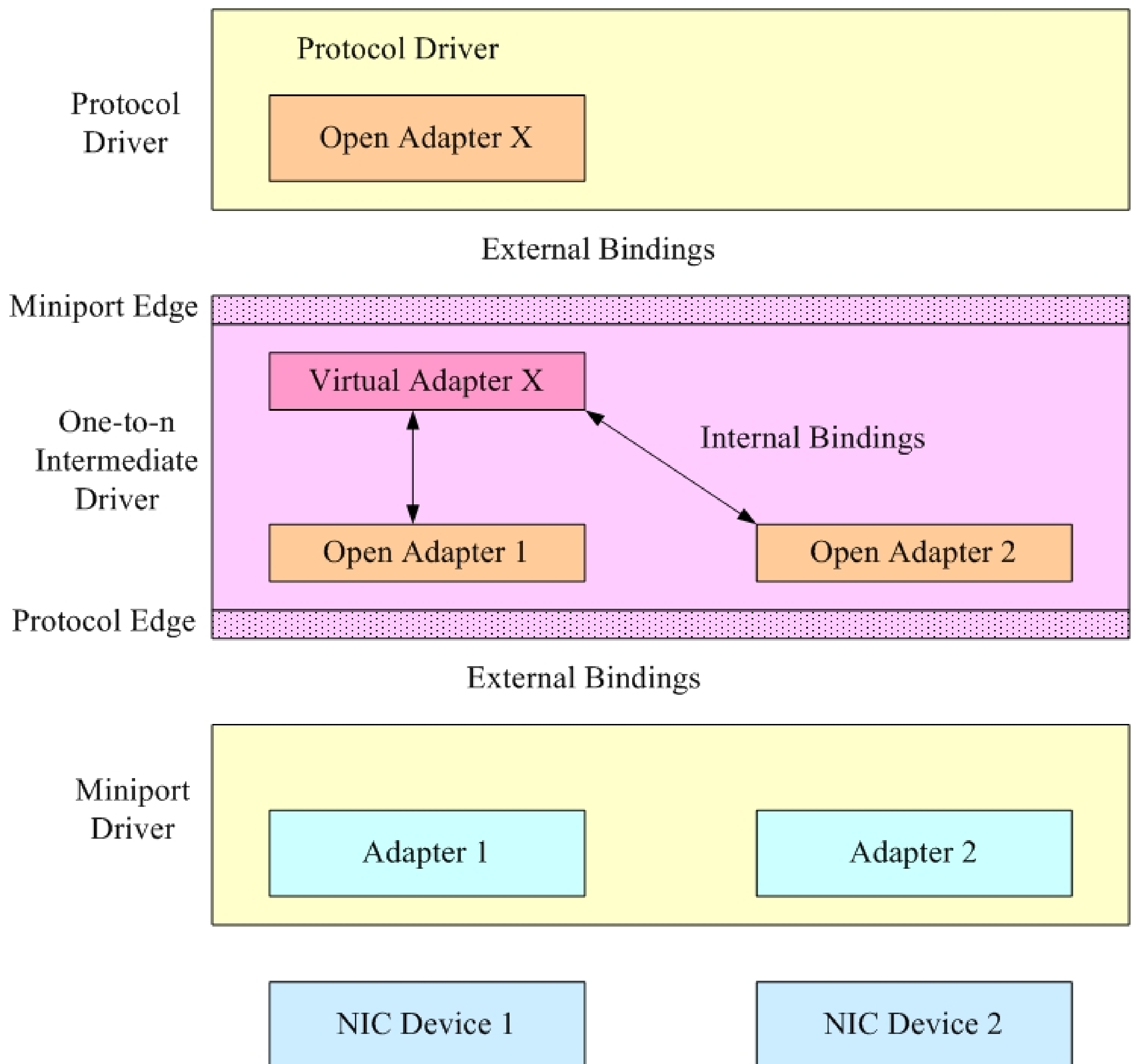



圖 2-9 一對多過濾型中介層驅動程式

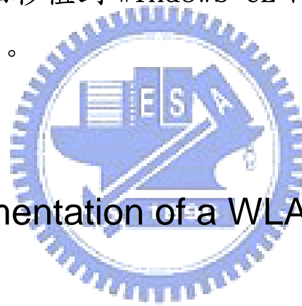
# 第三章 相關論文研究

## 3.1 無線異質網路的整合

在這個節，我將介紹幾篇和無線異質網路整合相關的論文和實作系統。

### 3.1.1 異質網路漫遊系統整合平台之設計與實作

這是由交通大學資訊工程系無線網際網路實驗室范榮軒同學所提出的論文，在論文中，范同學在 Windows XP 作業系統上實作了一套 *RIOMIP* 系統，透過 Mobile IP 協定，能讓行動端能在異質網路間漫遊，而不會斷線。我的論文研究主要是基於這套系統，利用它的部份軟體元件，將整個移植到 Windows CE 作業平台上，並進一步的改進。我將在後面章節介紹 *RIOMIP* 系統。



### 3.1.2 Design and Implementation of a WLAN/cdma2000 Interworking Architecture

在這篇論文中，介紹了著名的貝爾實驗室在它們的IOTA (Integration Of Two Access Technologies, IOTA)計劃下所開發的無線網路整合系統。其中包含兩個部份，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 3-1 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。

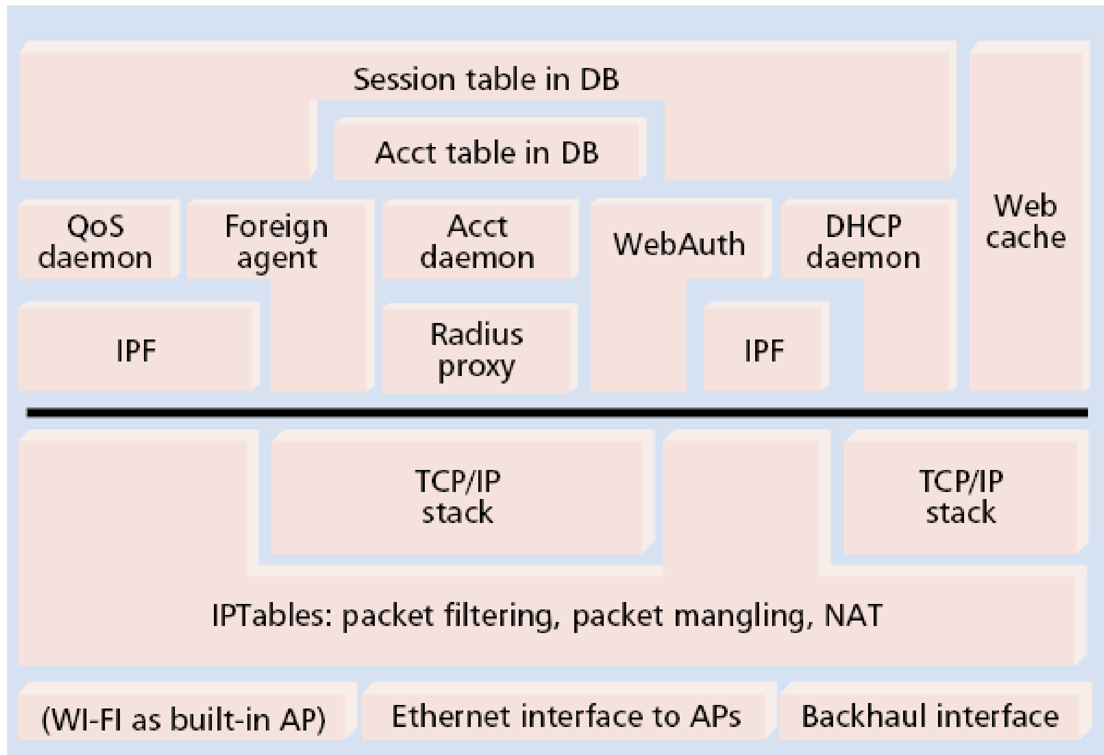



圖 3-1 IODATA 閘道器架構

- 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 3-2 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。

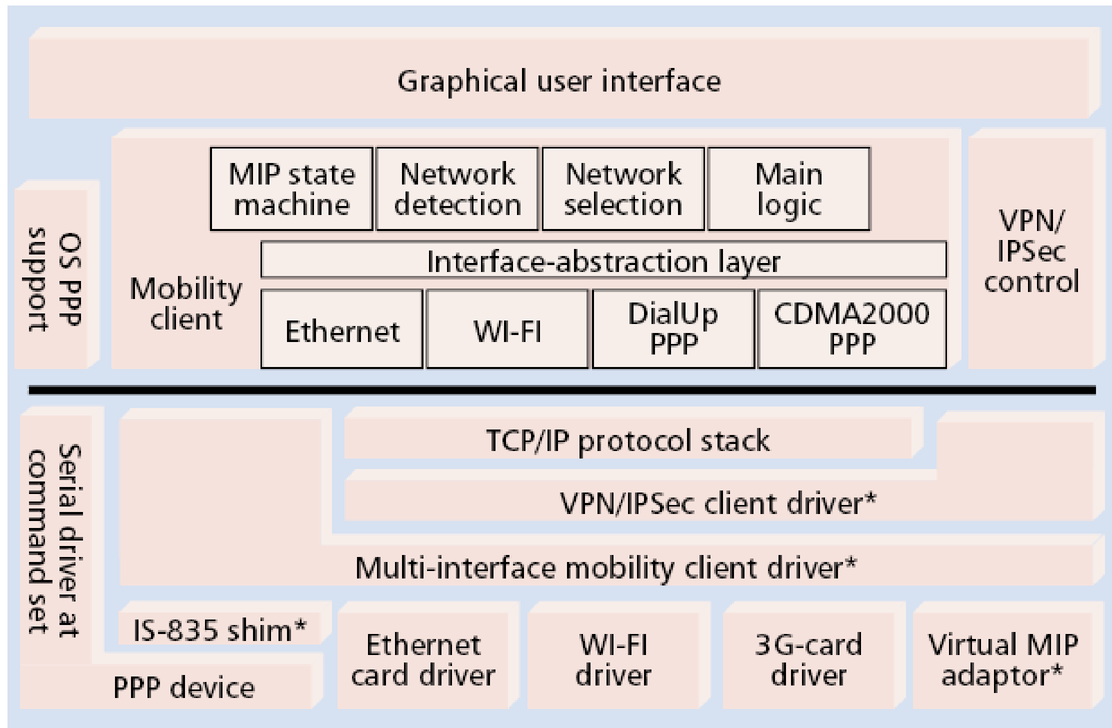



圖 3-2 IODATA 客戶端架構

## 3.2 跨階層網路設計

在這我將介紹寄篇討論跨階層網路設計概念的論文。

### 3.2.1 Cross-layer Design in 4G Wireless Terminal

在這篇論文中，指出傳統 TCP/IP 層疊式的架構造成了無線連結和行動端點上的效能不佳。在邁向第四代無線通訊網路時，作者提出了行動端點應該要有跨階層網路設計的概念。透過這樣的概念，行動端點能夠有較好的網路效能，較省點和較好的服務品質。為了闡述誇階層網路設計的好處，作者提出了 coordination planes 概念。作者定義所謂的 coordination planes 是指用一個跨階層的觀點來看協定堆疊，使得層級之間的合作演算法可以被套用。每個 coordination planes 都會注重解決某一類的問題。作者在論文中，提出了四種 coordination planes，如圖 3-3 所示:

- Security

  主要是為了解決多層級重複加密的問題，減少運算能量的浪費和傳輸效率的提升。

- QoS

在各層級間負責分配QoS的需求和限制，達到應用程式所需的服務品質。

- Mobility

解決行動端點在移動時所造成的網路效能低落的問題。例如Mobile IP換手時所造成的TCP congestion control以致效能低落的問題。

- Wireless Link Adaptation

解決因為無線網路連結特性所造成的問題。例如BER (Bit Error Rate)。

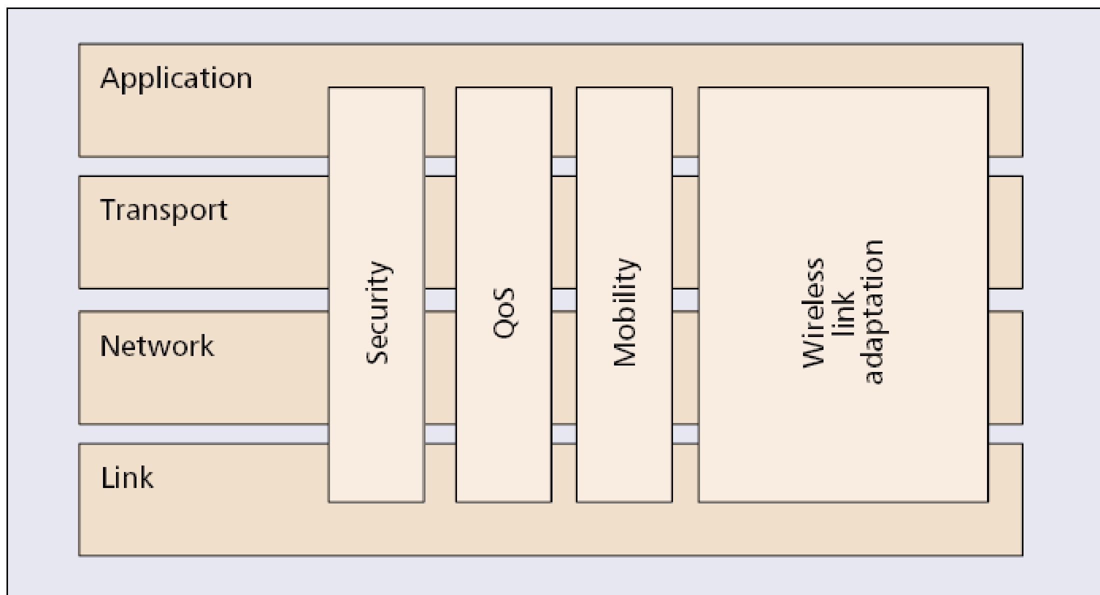



圖 3-3 CROSS-LAYER COORDINATION PLANES

根據上述的概念，作者提出一個可行的跨階層的合作模式，如圖 3-4 所示。不同層級的協定透過一個跨階層管理員模組，取得其他階層的狀態資訊，並回報該階層相關的事件。

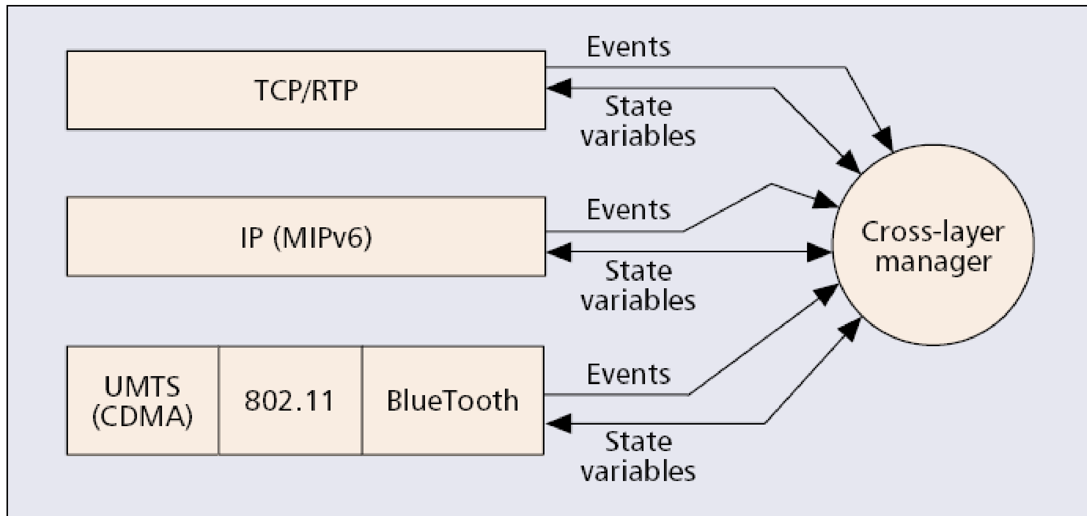



圖 3-4 跨階層合作模式

# 第四章 移植 RIOMIP

如第三章所敘述的，*RIOMIP* 為一能在多異質網路下進行漫遊的用戶端軟體。它主要是透過 Mobile IP 協定，在微軟視窗平台 （Windows 2000/XP）下提供切換使用各種異質媒介網路界面時的不斷線漫遊服務。本章將詳述如何將此軟體移植到 Windows CE .NET 為核心的作業系統平台上。

## 4.1 *RIOMIP* 軟體架構簡介

圖 4-1 所展示的便是 *RIOMIP* 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 CDMA2000 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。底下我將簡介各個元件的功能。

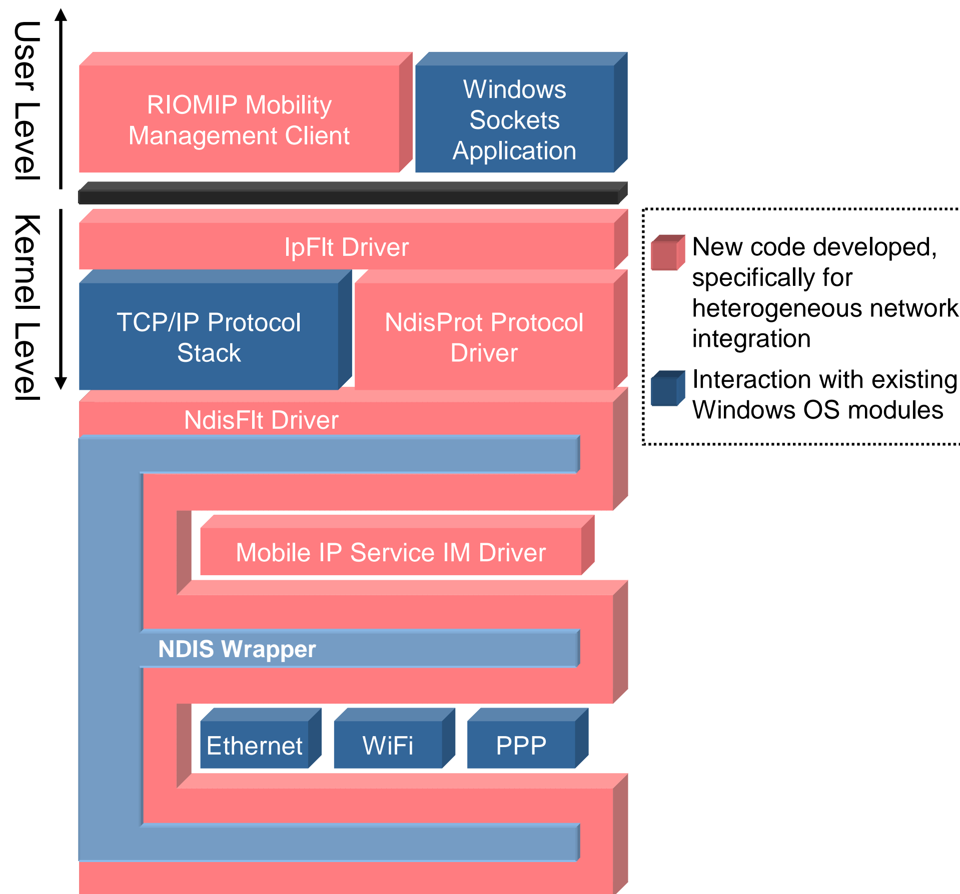



圖 4-1 *RIOMIP* 軟體架構圖

- NdisFlt Driver:

NdisFlt 驅動程式是利用所謂的 NDIS-Hook 的技巧來攔截封包用的驅動程式。基本上它會去修改 ndis.sys 的匯出表（Export Table）來置換 NDIS Wrapper 所提供的函式，達到攔截封包的功能。此外它還提供了一個介面，讓其他在核心的驅動程式可以註冊回叫函式（Callback Function），決定過濾封包的規則（Rule），做過濾封包的核心動作。

- IpFlt Driver:

IpFlt 驅動程式最主要是向 NdisFlt 驅動程式註冊過濾封包的回叫函式(Callback Function)。另外一個功能則是提供一個介面，來傳遞攔截到的封包給在使用者層級的 *RIOMIP* Mobility Management Client 中的封包監測模組（Packet Sniff Module）。

- 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，其用來偵測網路介面的媒體是否處於連結狀態。當媒體不除於連線狀態時，介面卡跟 TCP/IP 通訊協定的聯繫會被取消，會造成該網路介面卡的 TCP/IP 相關設定被移除，如 IP 位址與其相關的路由。此驅動程式就是來欺騙 TCP/IP 通訊協定驅動程式，來取消媒體感應的功能

- NdisProt Protocol Driver:

NdisProt 驅動程式為一協定驅動程式，但它的作用並非真正的實作某種特定的傳輸協定堆疊，而是用來提供一個介面作為用戶模式程式與核心模式 NDIS 驅動程式間溝通的橋樑。

- *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:

- NDIS 用戶模式輸入/出模組 (NDIS User-mode I/O Module): 負責與 NdisProt 協定驅動程式間的溝通。主要的功能為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。另一個重要功能則是直接往底層迷你連接埠驅動程式傳遞封包。
- 封包監視模組 (Packet Sniff Module): 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。
- 配接卡管理模組 (Adapter Management Module): 主要功能為負責對於網路卡的使用及切換。透過系統所提供的 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。此模組也會利用 NDIS 用戶模式輸入/出模組取得物件識別元以及網路媒體事件通知，作為網路卡切換之判斷依據。而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。

- 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。
- 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。
- 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。該介面還提供讓使用者可以手動強迫切換所使用的配接卡。

1896


## 4.2 移植課題

本節當中，我將敘述將 *RIOMIP* 移植到 Windows CE 平台上所遇到的問題。這些問題主要是因為作業系統架構的不同所引起，其次是系統函式庫的缺少或不相容。

### 4.2.1 應用程式和裝置驅動程式之間的輸入/輸出(I/O)方式

在 Windows NT 系列的作用平台上，所有的輸入輸出都是封包驅動（Packet Driven）的。每一個輸入輸出的動作，都會被一個封包所描述，藉由這個封包所帶的資訊來告訴驅動程式應該做些什麼動作。而這個封包被稱為 IRP（I/O Request Packet）。整個 I/O 運作的流程如約略為下面幾個步驟：

1. User-mode 的應用程式透過系統的 Win32 Subsystem 發出 I/O 請求，例如呼叫 CreateFile()、ReadFile()和 WriteFile()等系統函式。

2. 系統的輸入輸出管理員（I/O Manager）會針對這個 I/O 請求，配置出一個相對應的 IRP。

3. 輸入輸出管理員根據應用程式請求的 I/O 的功能和檔案的 Handle，將 IRP 傳遞給適當的驅動程式的 Dispatch-routine

4. Dispatch-routine 檢查 IRP 中所帶的請求的參數，如果合法，可選擇買上執行請求的動作，或者把這個 IRP 標記為待處理（Pending） ，必且把該 IRP 放到佇列裡面。

5. 把控制權交還給系統的輸入輸出管理員，輸入輸出管理員在交還給 User-mode 的應用程式，如果請求的 I/O 動作已經完成，執行的結果會一並傳回來。

整個流程如圖 4-3 所示。由於 Dispatch-routine 可以不馬上處理這個 I /O 請求，而先把控制權轉回給使用者應用程式。所以整個 I /O 的動作可以是非同步的(Asynchronous)。非同步 I /O 的好處就是，執行程序和 I/O 處理程序間可以重疊(overlapping)，執行程序可以不用等待 I/O 處理程序完成，繼續處理其他的工作。同時，這項特性也讓單一的執行程序可以同時執行多個輸入輸出的動作。同步和非同步的輸入輸出概念如下圖 4-2 所示。

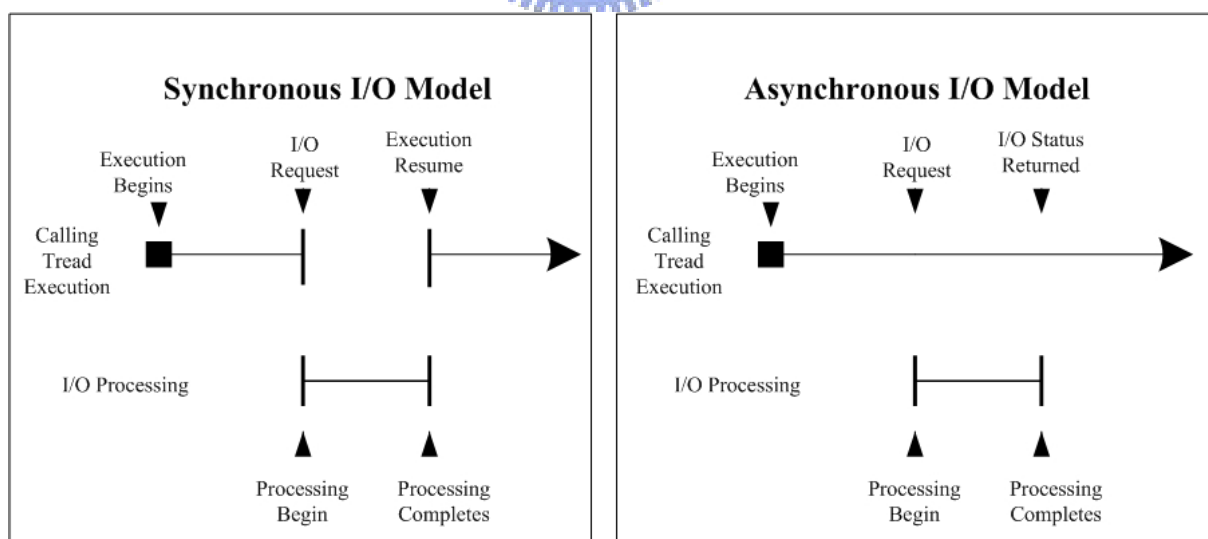



圖 4-2 SYNCHRONOUS I/O VS. ASYNCHRONOUS I/O

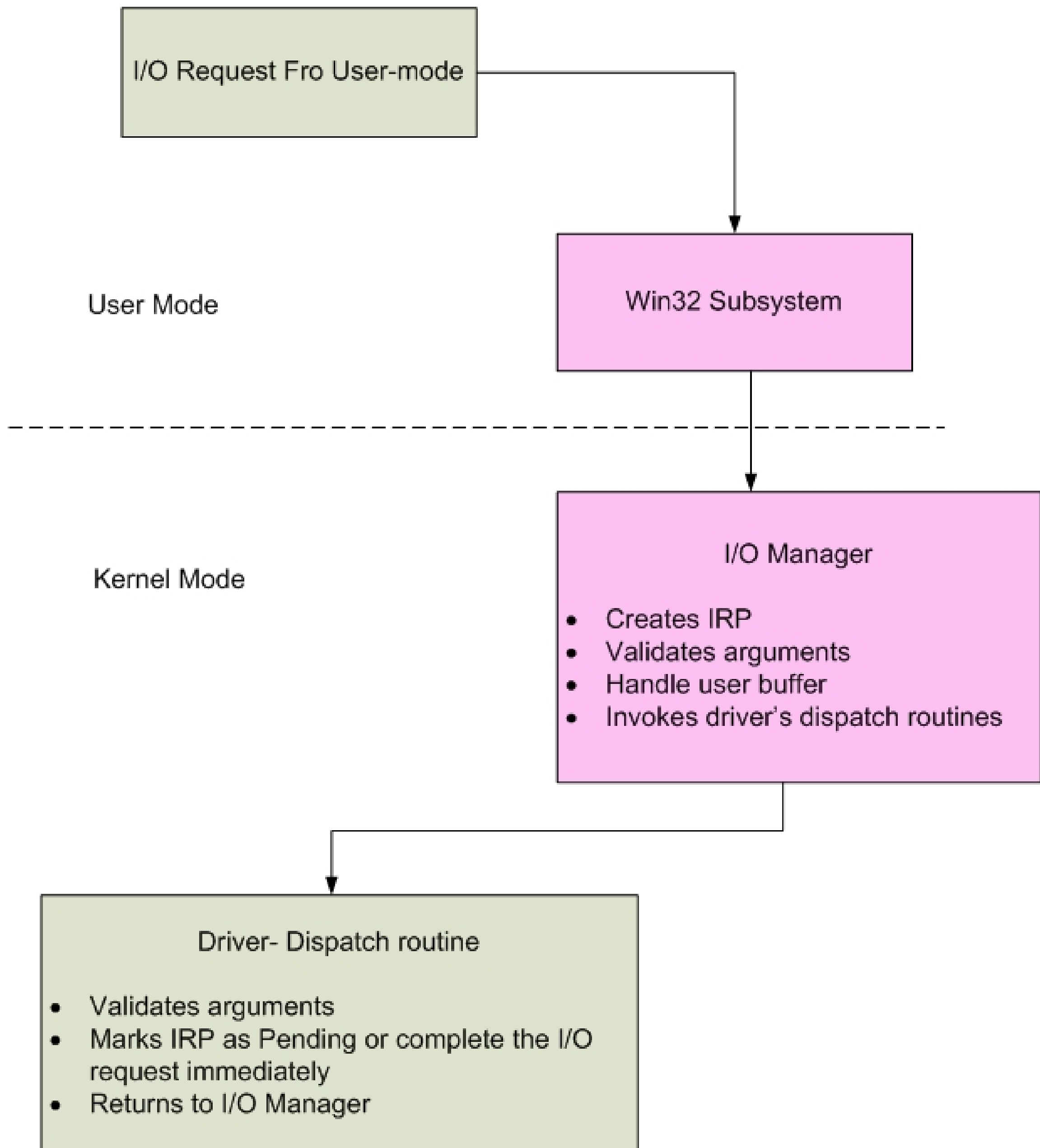



圖 4-3 應用程式與驅動程式 I/O 呼叫流程圖

而在 *RIOMIP* 當中，IpFlt 驅動程式要將過攔截到的封包要傳回給 Packet Sniff Module 時，就是利用上述這些非同步的特性。底下將敘述整個封包傳遞的流程：

1. Packet Sniff Module 透過呼叫多個 DeviceIoControl 函式，將欲存放封包的多個 Buffer 的位址傳送給 IpFlt Drvier。 如前面所述，系統的輸入輸出管理員會把這些 I/O Request 轉換成對應的 IRP，傳遞給 IpFlt Driver，其緩衝區的記憶體位址資訊也會存放在 IRP 當中。

2. IpFlt Driver 將這些輸入輸出請求的 IRP 標記為待處理(呼叫 MarkIrpPending 系統函式),必且將這些 IRP 放在存放在佇列裡面。

3. 當攔截到封包時,將封包存放在某一個 IRP 所記錄的 Buffer 裡面。

4. Packet Sniff Modulec 會用 GetOverlappedeResult 系統函式來等待輸入輸出動作的結束。

5. 當 Buffer 滿了之後,或者是一段時間結束,IpFlt 驅動程式會去結束這個 IRP(呼叫 IoComplete 系統函式)來通知 Packet Sniff Module 整個 I/O 動作結束,讓 Packet Sniff Module 去緩衝區裡面讀取封包作後續的處理。

6. 再次呼叫 DeviceIoControl 系統函式,傳送新的緩衝區位址給 IpFlt 驅動程式。

整個流程如下圖 4-4 所示。而程式碼的呼叫流程則如圖 4-5。

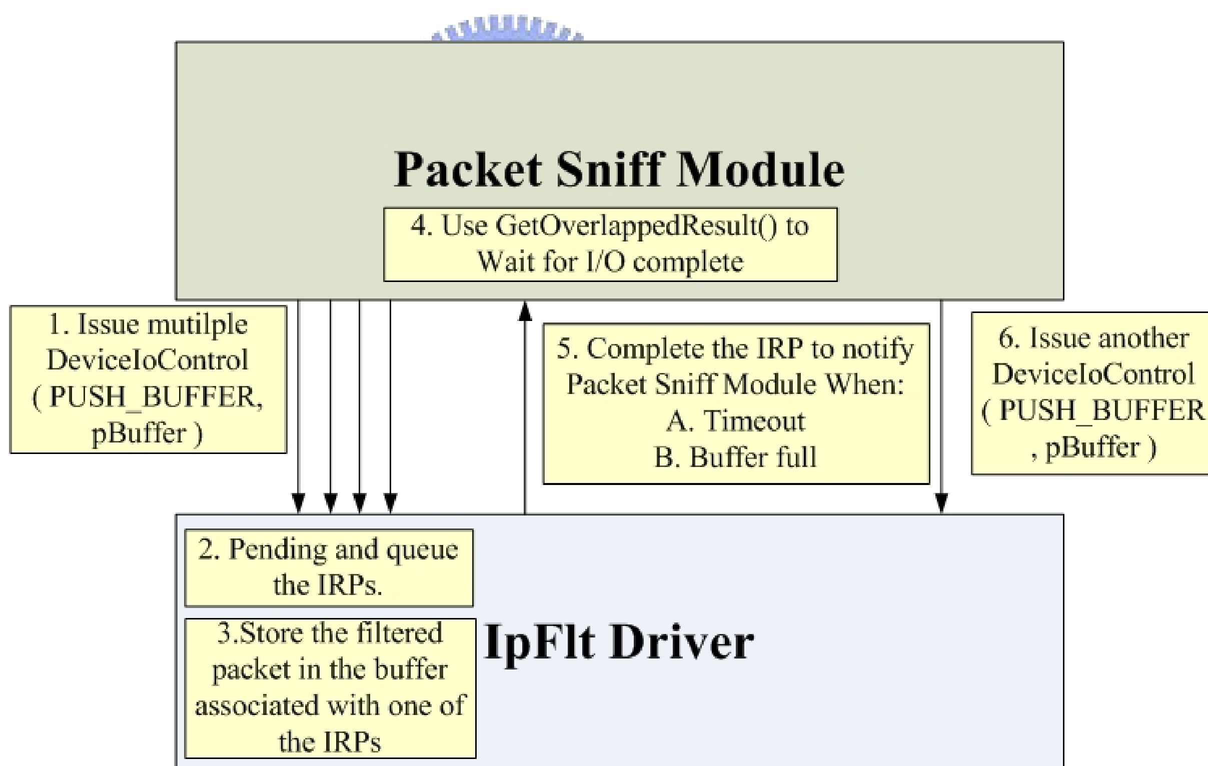



圖 4-4 *RIOMIP* 封包傳送流程圖

然而 Windows CE 作業平台上卻缺乏了這樣的機制。Windows CE 沒有 IRP 這種封包驅動的架構概念,所以也不支援非同步的輸入輸出方式。所以為了讓整個 *RIOMIP* 軟體移植過來,我將自己設計和實作應用程式和裝置驅動程式之間非同步的輸入輸出方式。

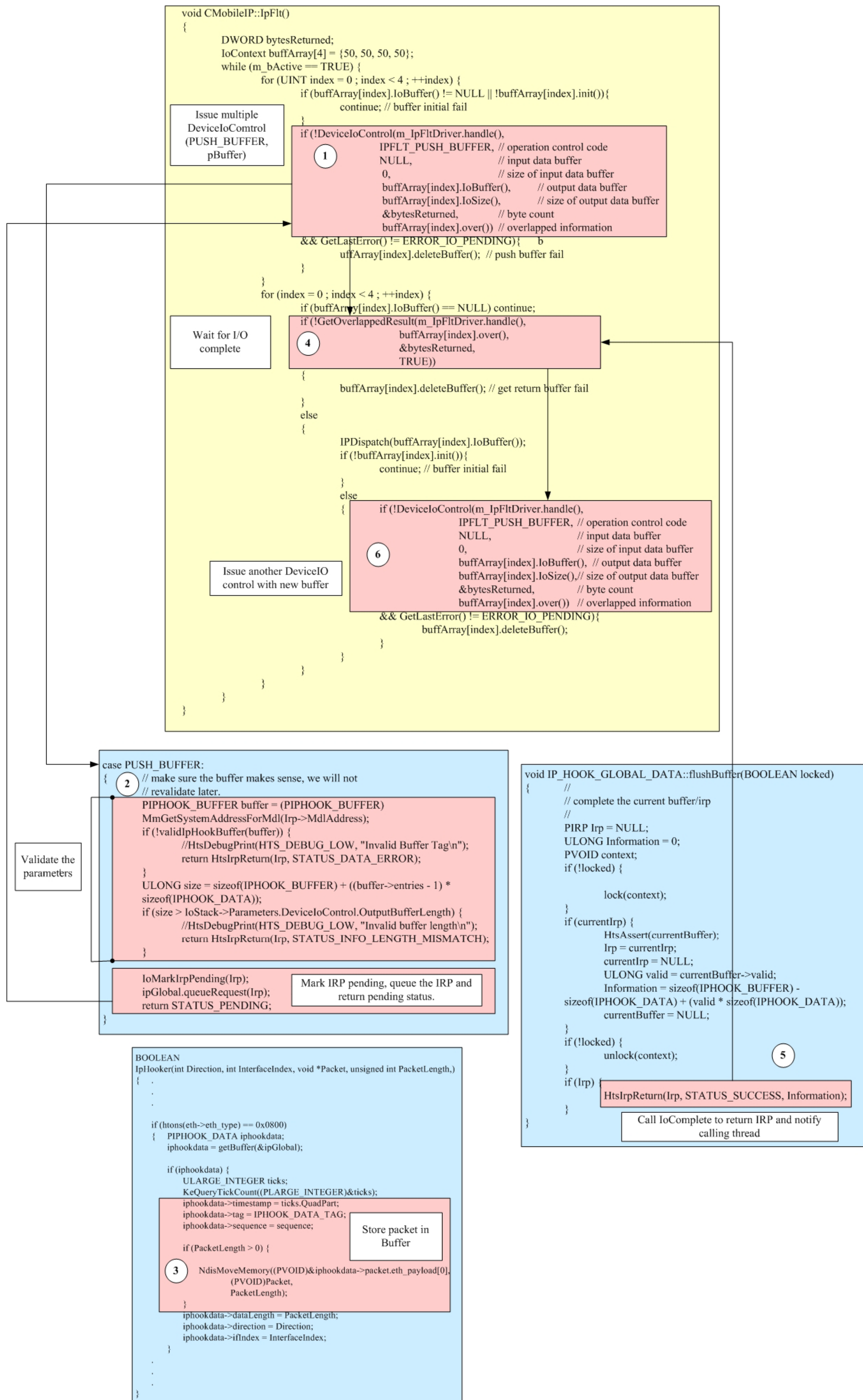



圖 4-5 封包傳送程式碼流程對應

### 4.2.2 封包攔截方式

在 *RIOMIP* 中，NdisFlt 驅動程式利用 NDIS 攔截（NDIS Hooking）的技巧來攔截封包。基本上，其運作原理是藉由修改 ndis.sys 的匯出表(Export Table)，來置換 NDIS 介面包裝所提供函式庫函式的位址，來指向我們自己所設定的函式。如此一來 ，所有呼叫 NDIS 的函式，都會先經過我們自己函式的處理。其架構示意圖如圖 4-6。

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案格式(Portable Executable File Format)。所有向其他的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。而主要要置換的函式有:

- NdisRegisterProtocol
- NdisDeRegisterProtocol
- NdisOpenAdapter
- NdisCloseAdapter

在置換上述 API 之後，我們可以置換呼叫 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。

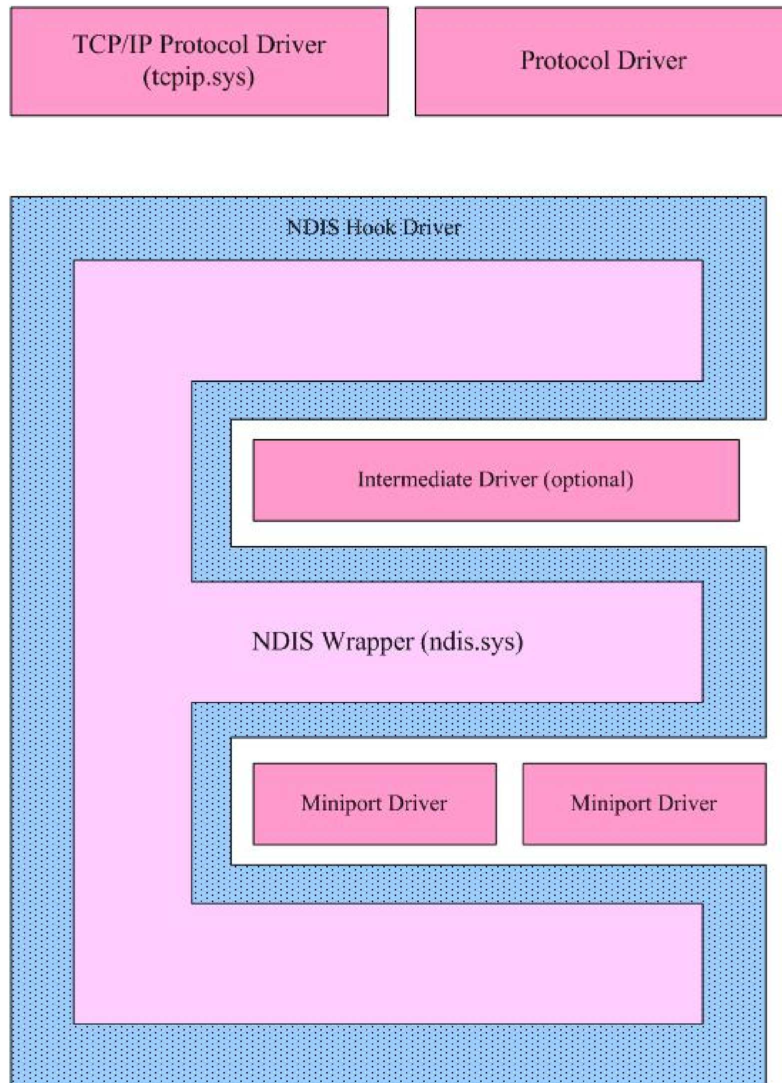



圖 4-6 NDIS HOOKING 架構示意圖

然而，這個方式微軟官方並不鼓勵使用，也沒有官方文件的支援(undocumented)，所以不利用跨平台的移植和日後版本更新的維護。所以我將採用一個同質性高，且符合微軟官方標準介面，實作一個 NDIS 中介層的驅動程式，來代替 NdsiFtl 驅動程式這個軟體元件。

### 4.2.3 應用程式和裝置驅動程式之間的輸入/輸出介面

前面 4.2.2 節中有提到，在 Windows NT 作業平台，所有應用程式對驅動程式間的輸入輸出動作，都是由 IRP 資料結構描述的。所有的輸入輸出動作都會被裝置管理員轉換成對應的 IRP 資料結構，傳送給驅動程式。在 IRP 資料結構裡面，有個 MjaorFuncion

的欄位，記錄著這個輸入輸出動作的功能型態。而我們在驅動程式裡面，我們可以針對不同輸入輸出功能的 IRP 資料結構，撰寫相對應 dispatch routine 來實作輸入輸出的動作。表 4-1 列出了所有 IRP 資料結構可能的輸入輸出的功能型態，和對應的系統函式。也就是說當我們呼叫這些系統函式時，會產生相對應輸入輸出型態的 IRP 資料結構。其中 IRP_MJ_INTERNAL_DEVICE_CONTROL 是兩個驅動程時之間做控制用，並無對應的系統函式。

| IRP I/O Function Code | Win32 API Function |
|---|---|
| IRP_MJ_CREATE | CreateFile |
| IRP_MJ_CLOSE | CloseHandle |
| IRP_MJ_READ | ReadFile |
| IRP_MJ_WRITE | WriteFile |
| IRP_MJ_DEVICE_CONTROL | DeviceIoControl |
| IRP_MJ_INTERNAL_DEVICE_CONTROL | X |

表 4-1 IRP 輸入輸出的功能型態

而我們在驅動程式的進入點函式當中，可以註冊位每個不同型態的 IRP 資料結構註冊相對應的 dispatch routine，如圖 4-7 所示。

```
CPP_DRIVER_ENTRY(PDRIVER_OBJECT DriverObject,
            PUNICODE_STRING RegistryPath)
{
    .
    .
    .
    DriverObject->MajorFunction[IRP_MJ_CREATE]          = IpHookCreate;
    DriverObject->MajorFunction[IRP_MJ_CLOSE]           = IpHookClose;
    DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL]  = IpHookDeviceControl;
    DriverObject->DriverUnload                          = IpHookUnload;
    .
    .
    .
}
```

圖 4-7 DISPATCH ROUTINE 註冊

而在 Windows CE 作業平台，並沒有 IRP 資料結構的概念，所以我們並不能用這樣的方式來實作應用程式和裝置驅動程式之間的輸入/輸出動作。然而在 Windows CE 作業系統裡面，定義了一種串流介面驅動程式 (Stream Interface Driver)，透過實作這型態的驅動程式介面，我們也可以實作應用程式和裝置驅動程式之間的輸入/輸出介面。關於串流介面驅動程式以及如何利用它來實作應用程式和裝置驅動程式之間的輸入/輸出介面我將在 4.3.3 節中詳細介紹。

### 4.2.4 動態主機設定協定客戶端函式庫（DHCP Client API）

在 Windows XP 作業平台中，系統提供了動態主機設定協定客戶端的函式庫。提供程式開發者利用系統的動態主機設定協定客戶端元件去跟動態主機設定協定伺服器查特定的資訊。

在原本的 *RIOMIP* 程式中，主要利用了此函式庫去取得動態主機設定協定閘道器的位址，和註冊一個事件，來觀察閘道器的位址是否改變。然而在 Windows CE 平台並不提供這樣的一個函式庫。所以我將利用別的實作方式，來提供相同的功能。

## 4.3 替代方案的設計與實作

### 4.3.1 非同步輸入輸出

如 4-1-1 所描述， Windows CE 作業平台並不支援非同步輸入輸出存取方式，所以我將自己設計和實作這項功能。

非同步輸入輸出的主要功能，就是可以讓使用者應用程式在呼叫輸入輸出函式返回之後，驅動程式再把資料寫回。同時驅動程式也能夠告知使用者應用程式輸入輸出動作已經完成，資料已經寫回，讓使用者應用程式能夠適時的去讀取資料。

在介紹我的實作方式之前，底下我將先介紹 Windows CE 作業平台上面記憶體管理和保護的機制，來闡明實作非同步輸入輸出存取方式時，所會遇到的困難和解決方式。

在 Windows CE 的環境中，對於所有的應用程式而言，所看到的記憶體空間為 2 GB 的單一虛擬記憶體空間（single virtual address space），如圖 4-8 所示。其中一部分的記體空間被劃分成 33 個 32 MB 的記體槽（slot）。其中 2 到 33 號的記憶體槽會被

分配給每個不同的應用程式的程序（process）；而 0 號記體槽則保留給現在正在執行的應用程式，1 號的記憶體槽則是保留給 Execute- In-Place（XIP）的 DLL。0 號和 1 號記體體槽可以視為單一 64-MB 的應用程式記憶體空間。

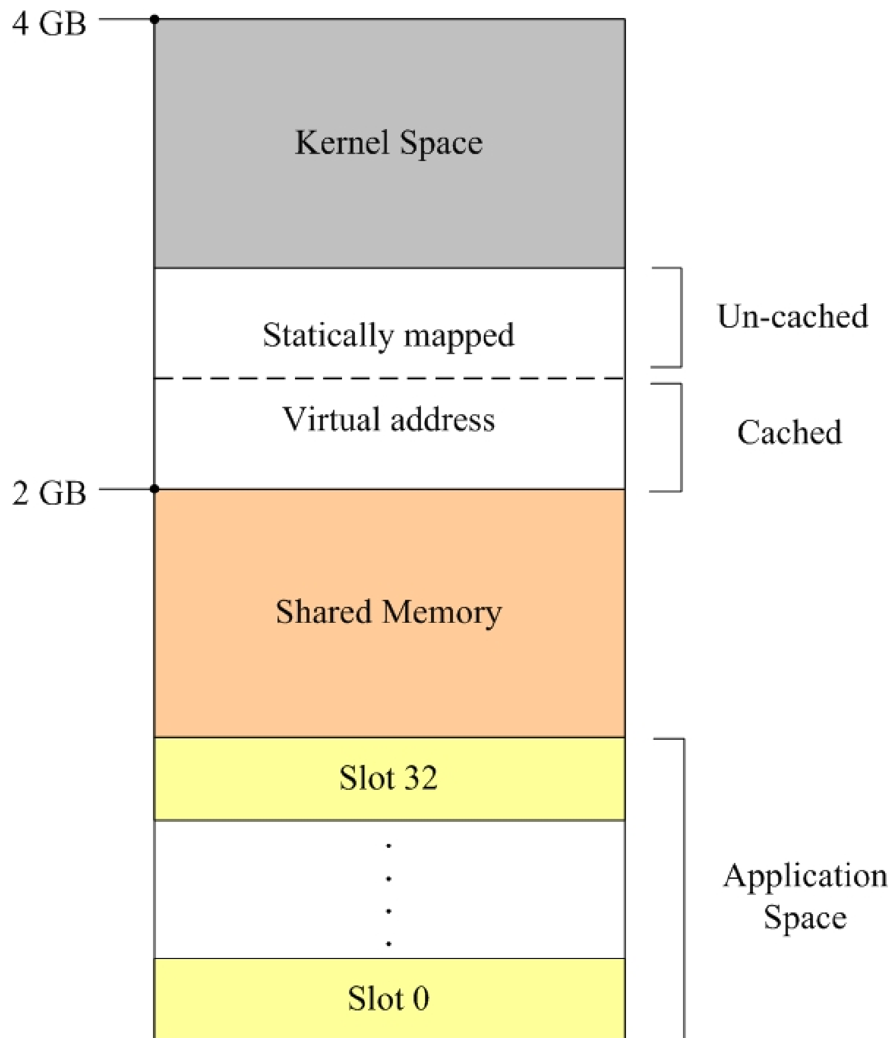



圖 4-8 WINDOWS CE 虛擬計憶體分配圖

當一個程序的執行緒在執行時，擁有這個執行緒的程序的記憶體槽將被複製（clone）一份到 0 號記憶體槽。準確一點來說，作業系統是去更改分頁表，讓這兩個記憶體槽的虛擬記憶體位址指向同一個實體記憶體位址，即該程序所屬的記憶體槽所對應的實體記憶體位址。而每個應用程式只能存取自己的記憶體槽，存取別人的記憶體槽將會發生記憶體保護例外（Memory protection exception）。所以說，雖然 Windows CE 作業系統所實作的是單一的虛擬記憶體空間，但是藉由這樣的機制，提供了記憶體的保護機制。

當應用程式呼叫系統函式時，作業系統中處理這些系統函式的程序，如 NK，FileSys，Device 或 GWES，會被允許存取呼叫系統函式的程序所屬的記憶體槽。所以當驅動程式在處理像是 ReadFile，WriteFile 或 DeviceIoControl 這些對他操作系統函式時，可以去寫入資料到屬於呼叫程序的記憶體槽的緩衝區。但是一旦這些函式完成，驅動程式將會失去權限去存取呼叫程序的記憶體槽。

所以，當我們要實作非同步的輸入輸出時，就會產生問題。因為在非同步輸入輸出的狀況下，驅動程式是在系統的輸入數出函釋返回後，驅動程式再把資料寫回，但是此時驅動程式必沒有權限將資料寫入該記憶槽。

為了解決這個問題，我們必須使用到一個特殊的系統函式，該函式可以更改記憶體的存取權限。該函式為 SetProcPermssions，而他的原型宣告為：

```
DWORD SetProcPermissions (DWORD newperms);
```

傳入的參數為一位元對應（bitmap），每一個位元代表是否擁有存取相對記憶體槽的權限。例如，最低的那個位元為 1 的話，代表呼叫該函式的程序擁有存取第一號記憶體槽的權限。回傳值則代表，呼叫該函式前程序所能存取的記憶體槽的位元對應。不過要注意的一點，記憶體保護的機制是為了讓程式設計師減少犯錯的機會，雖然我們藉由這個函式可以存取任意的記憶體槽，但仍需要小心使用。微軟的文件中特別指出，不應該隨意的把記憶體槽存取權限的位元對應設定成 0xFFFFFFFF，讓呼叫的程序可以存取所有系統中的記憶體槽。

另外，為了查詢現在程序所擁有的存取權限，我們還可以呼叫下面這個系統函式:

```
DWORD GetCurrentPermissions(void);
```

這個函式會回傳目前程序的記憶體槽存取權限的位元對應。如果我們在驅動程式呼叫這個系統函式，得到的存取權限會包括裝置管理員，和呼叫系統函式正在存取驅動程式的程序。

另一個問題是：當應用程式配置了一個緩衝區時，這個緩衝區會位於 0 號記憶體槽。而當應用程式把指向這個緩衝區的指標當作參數傳遞給驅動程式，問題就發生了。因為驅動程式是由裝置管理程序（device.exe）所載入的，所以當驅動程式接收到輸入輸出

存取的呼叫時，驅動程式和裝置管理員都會被載入到 0 號記憶體槽。所以應用程式所傳遞過來的緩衝區指標將不再合法，因為指標所指向的緩衝區以不存在 0 號記憶體槽。

如果這個指標是系統函式中的參數，作業系統會自動幫你對應到呼叫該系統函式的程序所屬的記憶體槽。因為任何應用程式在 0 號記憶體槽所配置的緩衝區，也會在被配置在自己的記憶體槽。關鍵在於，當一個應用程式在執行時，他會從他所屬的記憶體槽被複製一份到 0 號記憶體槽，而這個複製的過程如同前面提到的，其實是複製分頁表中的記憶體對應，讓兩個記憶體槽指向同樣的實體記憶體位址。這樣一來，執行中的應用程式在 0 號記憶體槽所做的任何配置動作，都會反映到他原本所屬的記憶體槽。

作業系統會幫你把系統函式參數中的指標對應，但是如果這個指標是指向一個資料結構，然而這個資料結構中又有另外一個指標指向其他緩衝區，作業系統並沒辦法幫這個資料結構中的指標做轉換的動作。所以我們必須自己來做這樣的對應動作。系統提供了下面的函式來做指標的對應的動作：

```
LPVOID MapPtrToProcess(LPVOID ptr, HANDLE hProc);
```

其中第一個參數式要對應的指標；第二個參數則是指標所指向的緩衝區所屬的程序。而驅動程式要的知道呼叫他的程序，則可以使用下列的函式：

```
HANDLE GetCallerProcess(void);
```

一般來說，上面者兩個函式在可以在驅動程式中組合來使用，如下：

```
pMapped = MapPtrToProcess(pIn, GetCallerProcess());
```

在能夠把資料寫回應用程式的緩衝區後，另一個問題就是，驅動程式如何告知應用程式資料已經寫回去。我們在這邊是使用所謂共享事件（shared event）的方式。事件是 Windows 作業系統中的一種物件，它可以讓兩個執行緒作彼此間的同步動作，所以事件也被稱為一種同步物件。然而事件通常是在同一個程序中所使用的，如果要在不同程序中共用同一個事件，我們就稱為共享事件。因為應用程式和驅動程式屬於兩個不同的程序，所以我們必須利用共享事件的方式。

一般事件的運作模式如下：

1. 呼叫 CreateEvent 來建立事件 A。

2. 執行緒 B 必須等待執行緒 A 做完工作 A 後，執行緒才繼續執行。所以執行緒 B 呼叫 WaitForSingleObject 來等待事件 A 的發生。

3. 執行緒作完工作 A 之後，呼叫 SetEvent 來觸發 A 事件，來告之執行緒 B。

4. 執行緒 B 被喚醒，呼叫 ResetEvent 來重設事件 A

整個的流程如圖 4-9 所示。

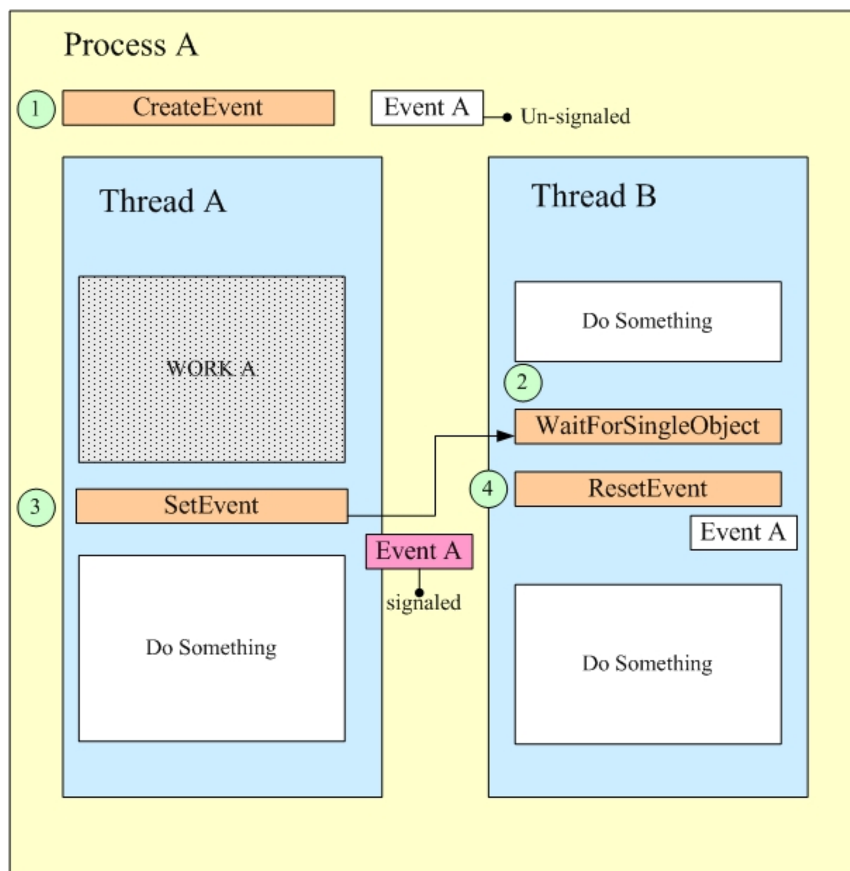



圖 4-9 事件運作模式

在上面的例子當中，兩個執行緒都在同一個程序裡面，使用了相同的記憶體空間，所以所有的全域變數都是可以共用的。然而如果要在兩個不同程序去共用一個事件，例如在我們的情況當中，需要應用程式和驅動程式去共用一個事件，則會像是下面的流程：

1. 應用程式的程序建立事件 A

2. 利用 DeviceIoControl 將事件的 Hanlde 傳給驅動程式。

3. 驅動程式呼叫 DuplicateHandle 複製一個新的 Handle 同樣指向事件 A。

4. 應用程式的執行緒等待事件 A 的發生。

5. 驅動程式觸發事件 A 來告知應用程式。

6. 應用程式重設事件 A，把事件 A 還原成未觸發狀態。

整個的流程如圖 4-10 所示。

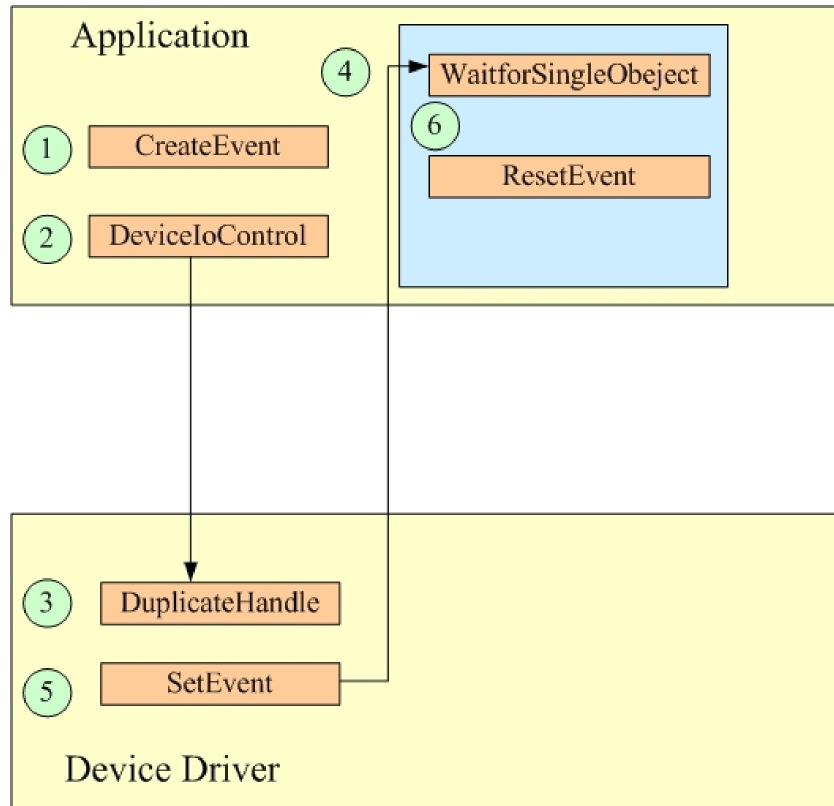



圖 4-10 共享事件運作模式

綜合上面所述，為了實作非同步的輸入輸出，我在我的驅動程式裡定義了如下的資料結構，來模擬在 Windows XP 作業平台上 IRP 的功能。

```
typedef struct _ASYNC_IO    {

      HANDLE                ayncEvent;
      PIPHOOK_BUFFER        buffer;
      DWORD                 dwCurrPermissions;

} ASYNC_IO, *PASYNC_IO;
```

ASYNC_IO 的資料結構裡面，除了紀錄資料的緩衝區外，另外包含了兩個欄位。一個是 asyncEvent，用來存放輸入輸出動作完成所要觸發的非同步事件；另一個則是 dwCurrPermissions，用來存放寫入緩衝區時所需要的存取權限。

另一方面，針對應用程式方面，我將用來記錄輸入輸出動作的類別 IoContext 作了小幅度的修改。原本用來支援非同步輸入輸出的資料結構 OVERLAPPED 型態，在 Windows 作 CE 平台並不支援，所以我將使用一個事件物件來替代，用來代表非同步輸入輸出完成的事件。修改後的 IoContext 如下：

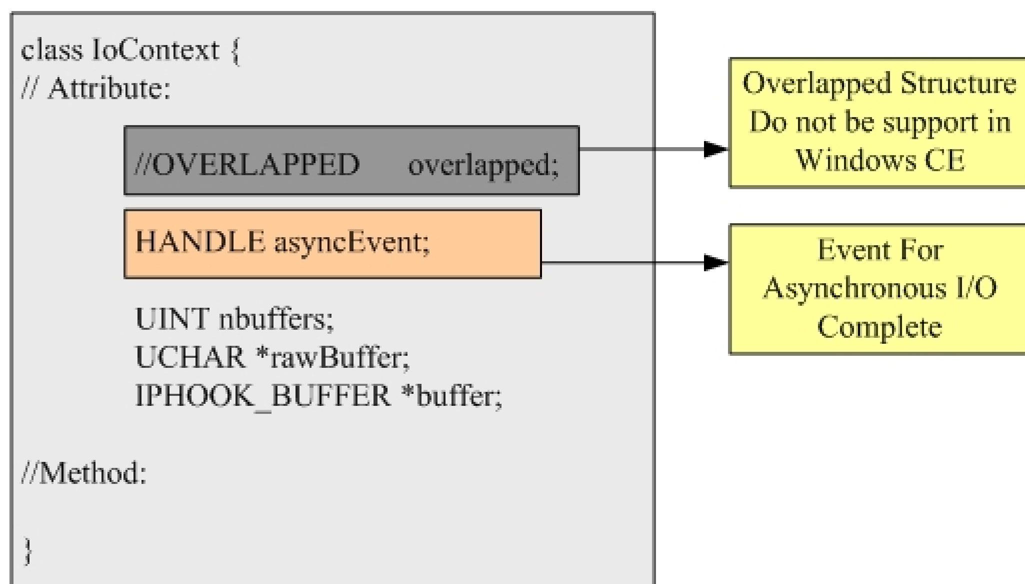



圖 4-11 IoContext 資料結構修改示意圖

而克服了非同步輸入輸出的問題後，經由小幅度的修改，原來的 Packet Sniff Module 就可以從驅動程式取得封包的資料。其運作的流程架構和溝通的介面幾乎是一樣的。我們從圖 4-12 可以看到，整個流程和之前的圖 4-x 相當類似，除了每個步驟中有細微的不同之外，而這些不同即是為了能在 Windows CE 作業系統上實現非同步的輸入輸出所作的額外動作。

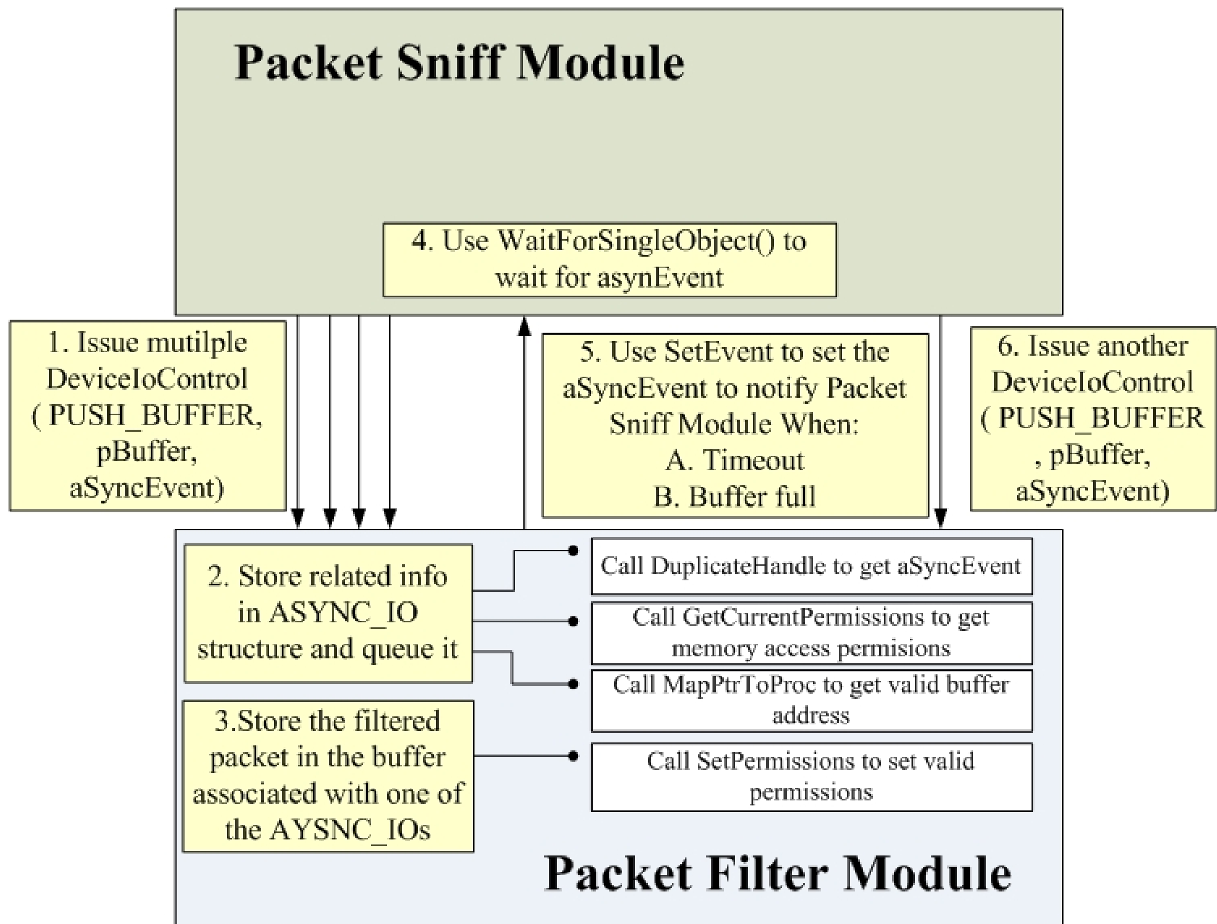



圖 4-12 封包傳遞流程示意圖

底下將敘述修改後封包傳遞的流程：

1. Packet Sniff Module 透過呼叫多個 DeviceIoControl 函式，將欲存放封包的多個 Buffer 的位址傳送給底層的驅程式。另 外，還必須把每個緩衝區相對應的非同步輸入輸出的事件，一並傳給驅動程式。

2. 驅動程式將這些實現非同步輸入輸出所需要的資訊，存入 ASYNC_IO 資料結構裡面。其中包括，利用 DuplicateHandle 系統函式來取得非同步輸入輸出事件；呼叫 GetCurrentPermissions 來取得當時合法的記憶體存取權限；另外，因為傳過來的緩衝區的資料結構中（圖 4-13），包含了指標指向真正儲存封包資料的緩衝區，所以必須做指標虛擬記憶體位址的轉換。

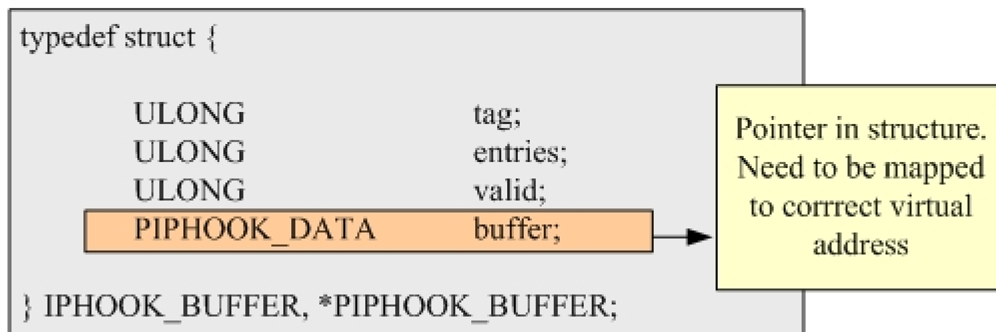



圖 4-13 緩衝區資料結構

3. 當攔截到封包時，將封包存放在某一個 ASYNC_IO 所記錄的緩衝區裡面。

4. Packet Sniff Module 會用 WaiForSingleObject 系統函式來等待驅動程式觸發非同步輸入輸出事件。

5. 緩衝區滿了之後，或者是一段時間結束，驅動程式會去呼叫 SetEvent 觸發非同步輸入輸出事件，讓 Packet Sniff Module 去緩衝區裡面讀取封包作後續的處理。

6. 再次呼叫 DeviceIoControl 系統函式，傳送新的緩衝區位址和相對應的非同步輸入輸出事件給驅動程式。另外因為非同步輸入輸出事件會重覆使用，所以需重設到未觸發的狀態。

相關重要程式碼和其流程如下圖 4-14 所示。

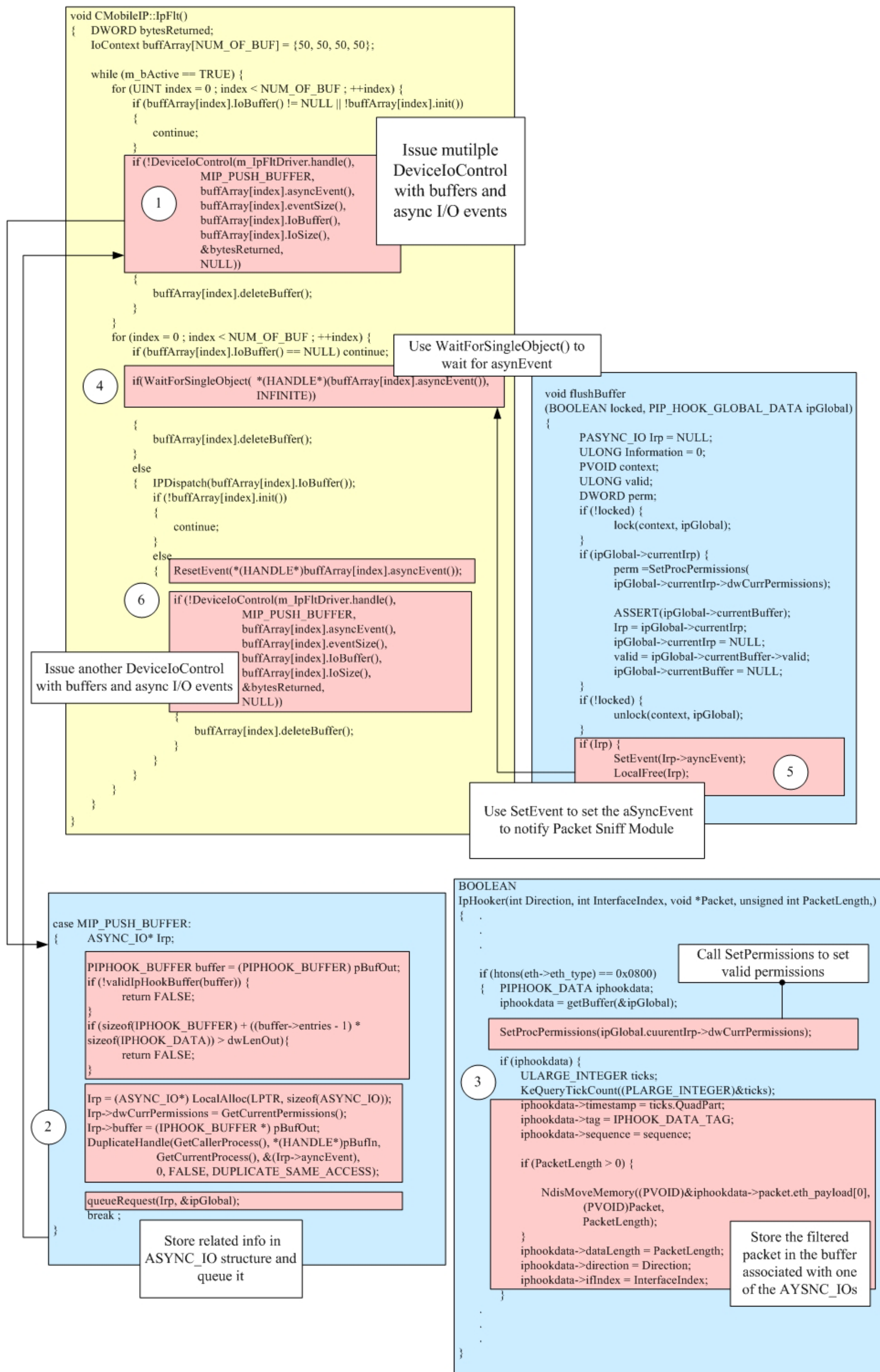



圖 4-14 封包傳送程式碼流程示意圖

### 4.3.2 網路中介層驅動程式是實作

在 2-2 中我曾經介紹過 NDIS 概念。其中有簡介了中介層驅動程式的概念和功能。藉由他能夠攔截封包的能力，我將實作一個中介層驅動程式來代替原來的 NdisFlt 驅動程式。在這節當中，我將詳細敘述如何實作一個網路中介層驅動程式，和如何利用它來做攔截封包的功能。因為功能的需求，我在這邊時作的是過濾型的中介層驅動程式。

要實作一個網路中介層驅動程式，必須要實作兩個介面：一個是協定驅動程式介面，一個是迷你連接埠驅動程式介面。前者就是所謂的 ProtocolXxx 函式，後者就是 MiniportXxx 函式。因為對上層的協定驅動程式來說，中介層驅動程式就好像迷你連接步趨動程式；而對下層的迷你連接埠驅動程式而言，中介層驅動程式則是扮演協定層驅動程式的角色。藉由這兩種介面，中介層驅動程式才能利用 NDIS 函式庫，和上下層的驅動程式互動。而這兩種介面，所需要實做的函式如表 4-2 所示，底下我就幾個重要的函式做詳細的說明。

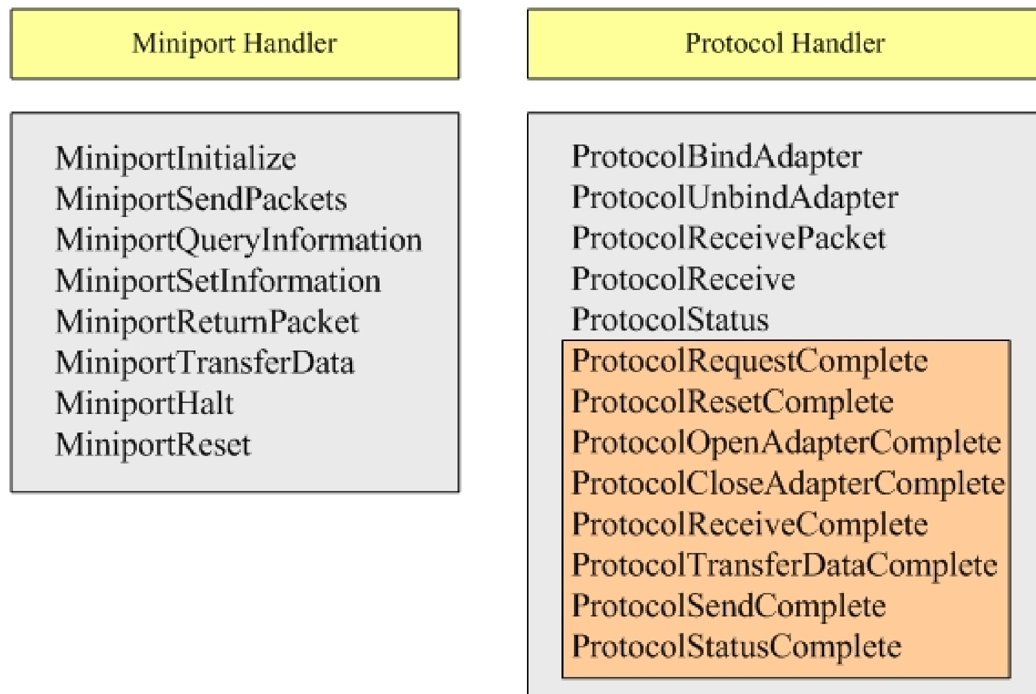



表 4-2 NDIS 中介層驅動程式介面函式

- 協定驅動程式介面

<u>ProtocolBindAdapter</u>

當每一個網路介面卡被迷你連接埠初始化後，NDIS 都會主動來呼叫這個函式，來要求協定驅動程式去跟底層的迷你連接埠驅動程式做連結。而在這個函式

裡面，最重要的是就是去呼叫 NdisOpenAdapter 去連結底層的網路界面裝置。然而，因為要實做中介層驅動程式，在這個函式裡必須去產生虛擬連接埠裝置。而透過呼叫 NdisIMInitializeDeviceInstanceEx 函式，我們可以產生一個虛擬連結埠。

而在過濾型的中介層驅動程式中，虛擬連接埠裝置根底層的實體網路介面卡的對應關係是一比一的，所以對於每個實體的網路介面卡，都必須產生一個虛擬連接埠裝置。也就是這個函式每被呼叫一次，就必須產生一個虛擬連接裝置，這和多路傳輸中介層驅動程式很不一樣的。

ProtocolRecievePacket

當底層的迷你連結埠驅動程式收到封包之後，它通常會呼叫 NdisMIndicateReceivePacket，而NDIS就會去呼叫與他有連結的協定驅動程式本處理函式，來做接收封包的動作。而在中介層驅動程式裡，在處理完封包之後，如果有需要，我們則必須再去呼叫NdisMIndicateReceivePacket，繼續把封包往上傳遞給上層的協定驅動程式。

ProtocolStatus

底層的迷你連接埠驅動程式通常會維護一些網路介面卡的狀態，當狀態有所改變時，它會去呼叫 NdisMIndicateStatus 來告知上層的協定驅動程式。而協定驅動程式就必須實做本處理函式，來接受這些狀態的改變。而在中介層驅動程式當中，我們通常會再次呼叫 NdisMIndicateStatus，來告知上層的協定驅動程式這些狀態的改變。

ProtocolSendComplete

由於網路傳輸的特性，為了得到較好的效能和降低中央處理器所浪費的時間,NDIS 支援上次的協定驅動程式和下層的迷你連接埠驅動程式之間的相互操作是可以非同步的。已傳送封包為例，在協定驅動程式呼叫了 NdisSend 之後，NDIS 函式庫會去呼叫底層迷你連接埠所實作的 MiniportSend 處理函式。但是，在 MiniportSend 處理函式中，迷你連接埠並不一定要馬上去傳送封包，因為某些效能考量和底層的網路介面卡的特性，MiniportSend 處理函式可以先回應 STATUS_PENDING，代表封包正在等待處理中，之後就返回。而剛剛協定驅動程式呼叫的 NdisSend 就會收到 STATUS_PENDING 的傳回執。而等到迷你連接埠真正的把封包透過網路介面卡傳送出去之後，它會去呼叫 NdisSendComplete 來告知協

定驅動程式剛剛的封包已經傳送出去。NDIS 函式庫就去呼叫協定驅動程式的 ProtocolSendComplete 處理函式，來做後續的動作。整個過程如圖 4-15 所示。通常我們會在這邊做資源釋放的動作，包括存放封包所用的緩衝區和封包描述子等。關於這些資源的詳細作用，我會在稍後討論到封包處理的部份，進一步的詳細說明

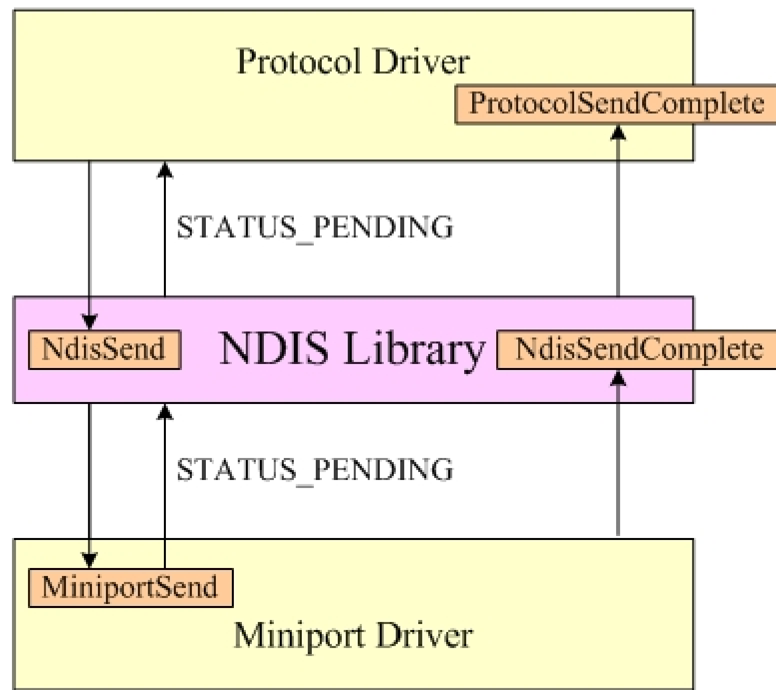



圖 4-15 NDIS 封包傳送非同步運作模式

- 迷你連接埠驅動程式介面

<u>MiniportInitialize</u>

在中介層驅動程式中，當協定驅動程式介面中的 Protocol- BindAdapter 函式被呼叫時，它會去呼叫 NdisIMInitialize- DeviceInstanceEx 函式來產生虛擬連接埠，然而 NDIS 函式庫就會呼叫本處理函式來作對虛擬函式庫作出初始話的動作。

<u>MiniportQueryInformation</u>

本處理函示負責接收由上層的協定驅動程式呼叫 NdisRequest 來傳送類別為 NdisRequestQueryInformation 的 OID_XXX 請求。這些請求是為了查詢下層迷你連接埠所維護的網路介面卡的狀況和網路參數設定。 除了一些特殊的情況，在中介

層驅動程式中，我們同常會在次呼叫 NdisRequest 函式把請求傳遞給下面的虛擬連接埠驅動程式。

MiniportSetInformationHandler

相對於 MiniportQueryInformation，這個函式接收的 OID_XXX 請求類別為 NdisRequestSetInformation，是用來設定下層迷你連接埠所維護的網路介面卡的參數。

MiniportSendPacket

在這個函式中，我們會接受到上層協定驅動程式所送下來的封包陣列。在這個函式中，我們需要重新包裝（repackage）封包，作配置新的封包描述子（packet descriptor)，串連封包的緩衝區等動作。最後在處理完封包之後，呼叫 NdisSend 或 Ndis- SendPackets 把封包傳送給底層的迷你連接埠驅動程式 。

進一步，我將介紹如何用過濾型中介層驅動程式來實作攔截封包的功能。首先，我將解說在 NDIS 驅動程式中封包是如何的被描述和儲存。

NDIS 封包結構主要是由兩種資料結構來表示：

- NDIS_PACKET: 所謂的封包描述子，用來描述在 NDIS 驅動程式裡面的一個封包，包含一指標來串連存放封包資料的緩衝區。底下則是 NDIS_PACKET 的完整資料結構。其中 Private 欄位唯一指標，用來串接用 NDIS_BUFFER 資料結構所表示的緩衝區；而 MiniportReserved 和 ProtocolReserved 為兩個保留欄位，在我們的中介層驅動程式當中，我們用這兩個欄位來存放指向上層協定驅動程式傳下來和底層迷你連接埠驅動程式傳上來的封包描述子的指標。資料結構如圖 4-16

```
typedef struct _NDIS_PACKET {
   NDIS_PACKET_PRIVATE  Private;
   union {
      struct {
         UCHAR      MiniportReserved[2*sizeof(PVOID)];
         UCHAR      WrapperReserved[2*sizeof(PVOID)];
      };
      struct {
         UCHAR      MiniportReservedEx[3*sizeof(PVOID)];
         UCHAR      WrapperReservedEx[sizeof(PVOID)];
      };
      struct {
         UCHAR      MacReserved[4*sizeof(PVOID)];
      };
   };
   ULONG_PTR          Reserved[2];
   UCHAR              ProtocolReserved[1];
} NDIS_PACKET, *PNDIS_PACKET, **PPNDIS_PACKET;
```

圖 4-16 NDIS PACKET 資料結構

- NDIS_BUFFER: 用來存放封包的緩衝區，在不同的平台，這個資料結構的型態都不一樣。在 Windows CE 作業系統中，NDIS_BUFFER 為一 MDL(Memory Descriptor List)，用來描述一段在虛擬記憶體空間連續的記憶體。

圖 4-17 和圖 418x 圖示上面兩種資料結構如何描述一個封包。須要注意的是，NDIS 並沒有規定一個封包需要用幾個 NDIS_BUFFER 來串連表示。

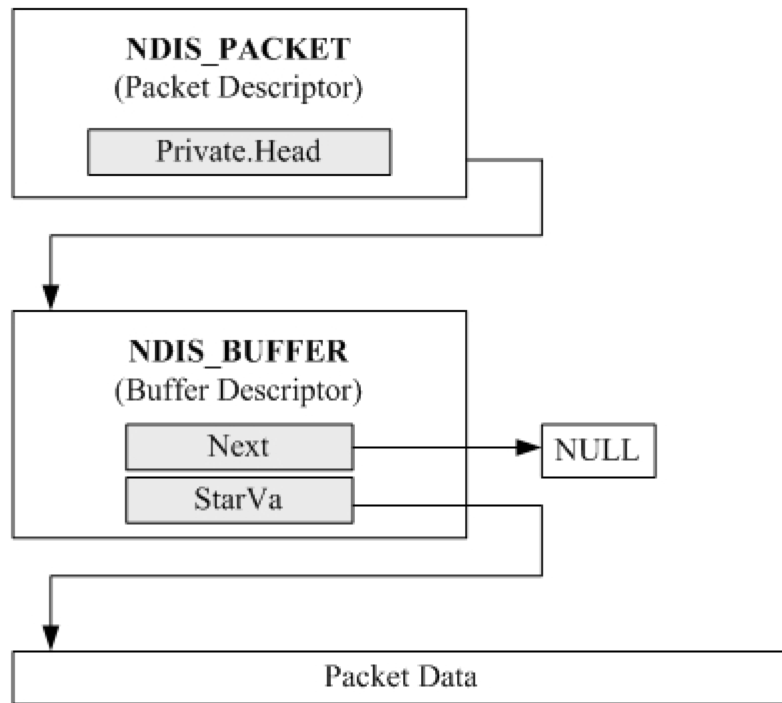



圖 4-17 SINGLE-BUFFER NDIS_PACKET 示意圖

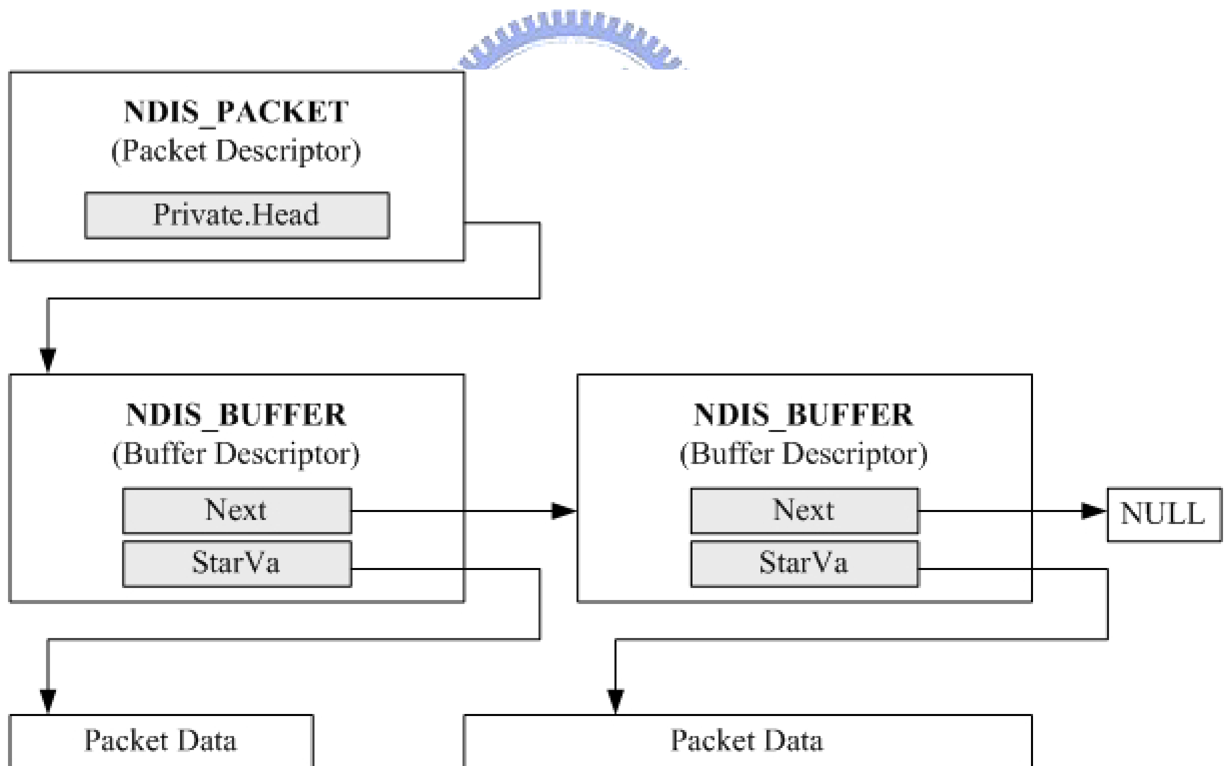



圖 4-18 MULTI-BUFFER NDIS_PACKET 示意圖

而在驅動程式裡面，我們通常並不直接存取這些資料結構的欄位，而是透過 NDIS 所提供的函式來存取封包。主要常用的函式和其功用如表 4-3 所示。

| NdisQueryPacket | Returns information about a given packet descriptor |
|---|---|
| *BufferCount* | Count of buffer descriptors chained to the packet descriptor |
| *FirstBuffer* | pointer to the initial buffer descriptor chained to the given packet descriptor |
| *TotalPacketLength* | The total number of bytes of packet data mapped by all chained buffer descriptors |
| NdisQueryBuffer | Returns information about a given buffer descriptor |
| *VirtualAddress* | Base virtual address of the virtual address range described by the buffer descriptor |
| *Length* | Number of bytes in the virtual address range described by the buffer descriptor. |
| NdisGetNextBuffer | Returns the next buffer descriptor in a chain, given a pointer to the current buffer descriptor |

表 4-3 NDIS 存取封包相關函式

在中介層驅動程式，不管是從上層協定驅動程式傳遞下來的封包，或者是由下層迷你連接埠驅動程式所上傳的封包，會是由 NDIS_PACKET 封包描述子所表示的。所以在程式中，我們必須藉由表 4-3 中的函式，才能存取到封包的內容。一般的步驟如下所示：

1. 呼叫 NdisQueryPacket 函式，我們可以得到封包描述子所串接的第一個 NDIS_BUFFER 緩衝區描述子。
2. 呼叫 NdisQueryBuffer 函式，可以取得步驟 1 所取 NDIS_BUFFE 緩衝區描述子的緩衝區的虛擬記憶體位址，即是一個指向封包資料的指標。
3. 因為每一個 NDIS_PACKET 封包描述子所串接的緩衝區可能不只一個，所以我們可以呼叫 NdisGetNextBuffer 來取得下一個 NDIS_BUFFE 緩衝區描述子，重複步驟 2，直到沒有傳接的緩衝區，就可以取得一個完整的封包內容。

底下圖 4-19 我實作的中介層驅動程式的片段程式碼，展示了如何取得的一個完整的封包內容。

然而，要做封包攔截的動作，其實就是在封包通過的路徑上，插入我們自己的過慮函式，透過上述的方法，取得封包的完整資料，檢查封包的內容，看這個封包是不是我們所要攔截的。如果是，就把封包的完整資料拷貝到特定的緩衝區，最後在利用前 4.3.1 節所敘述非同步輸入輸出的方法，把封包傳送給我們上層的應用程式模組。

而由之前所介紹的 NDIS 驅動程式界面，包括了迷你連接埠的介面和協定的介面，可以看出來外送和內收的封包都分別會經過 MiniportSendPackets 和 ProtocolRecievePackets 這兩個處理函式。

所以我們所要作的，就是把我們設計的過濾函式加入這兩個函式當中，把所要攔截的封包攔截下來；而對於其他的封包，則讓他繼續傳送給下層的迷你連接埠驅動程式或者是上層的協定驅動程式。

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

1       NdisQueryPacket(ndis_packet, NULL, &NdisBufferCount, &buffer, &packet_len);

        if ((buffer == NULL) || (packet_len < sizeof(struct ether_hdr)))
                return TRUE;

2       NdisQueryBuffer(buffer, &pointer, &buffer_len);

        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)
        {       .
                .
                .
                if (packet_offset < packet_len) {

                3       NdisGetNextBuffer(buffer, &buffer);

                        if (buffer == NULL) break;

                2       NdisQueryBuffer(buffer, &pointer, &buffer_len);

                        if (pointer == NULL || buffer_len <= 0) break;
                        buffer_offset = 0;
                }
        }
        .
        .
        .
        return result;
}
```

圖 4-19 取的封包內容範例

另外有一點很重要的是，因為不管從上層的協定驅動程式，或者是下層的迷你連接埠驅動程式所傳遞過來的封包描述子，都是屬於它們本身的。當我們要將封包繼續往上或往下傳送時，我們必須做一個重新包裝的動作。因為在封包描述子中有包含一些屬於原本驅動程式特別紀錄的資訊 (MiniportReserved 和 ProtocolReserved)。我們必須重新配置至一個新的封包描述子，將原本封包描述子的一些封包資訊複製一份到新的封包描述子。如果整個封包的內容並未作改變，你可以將新的封包描述子緩衝區的指標，指向原來封包描述子所指向的緩衝區描述子。但是由於原來封包描述子所指向的緩衝區描述子本身也是屬於原來的驅動程式的，所以我們對緩衝區描述子及所描述的緩衝區內容並不能任意更改。我們只能把它當作是唯讀的。如果我們要改變封包的內容，我們也比須配置屬於我們自己的緩衝區，和緩衝區描述子。

整個過濾封包和重新包裝封包的流程，如底下圖 4-20 所示。圖 4-20 的例子，是我實作的 MiniportSendPackets 函式的片段程式碼，是針對由上層協定驅動程式所傳遞下來的外送封包。由這個例子我們可以清楚的看出來整個運作的流程。首先，我過濾函式 filter_apcket 會先檢查這個封包是不是上層的應用程式模組想要的；如果是，它會回傳 FASLE，並且把封包的完整內容拷貝到特定的緩衝區；如果不是，她會回傳 TRUE，並讓後續的程式碼重新包裝封包，將封包往下面的迷你連接埠驅動程式傳送。相同的原理，針對由下層迷你連接埠所傳遞上來的內收封包，我在 ProtocolRecievePacket 函式的實作也是相同的作法。

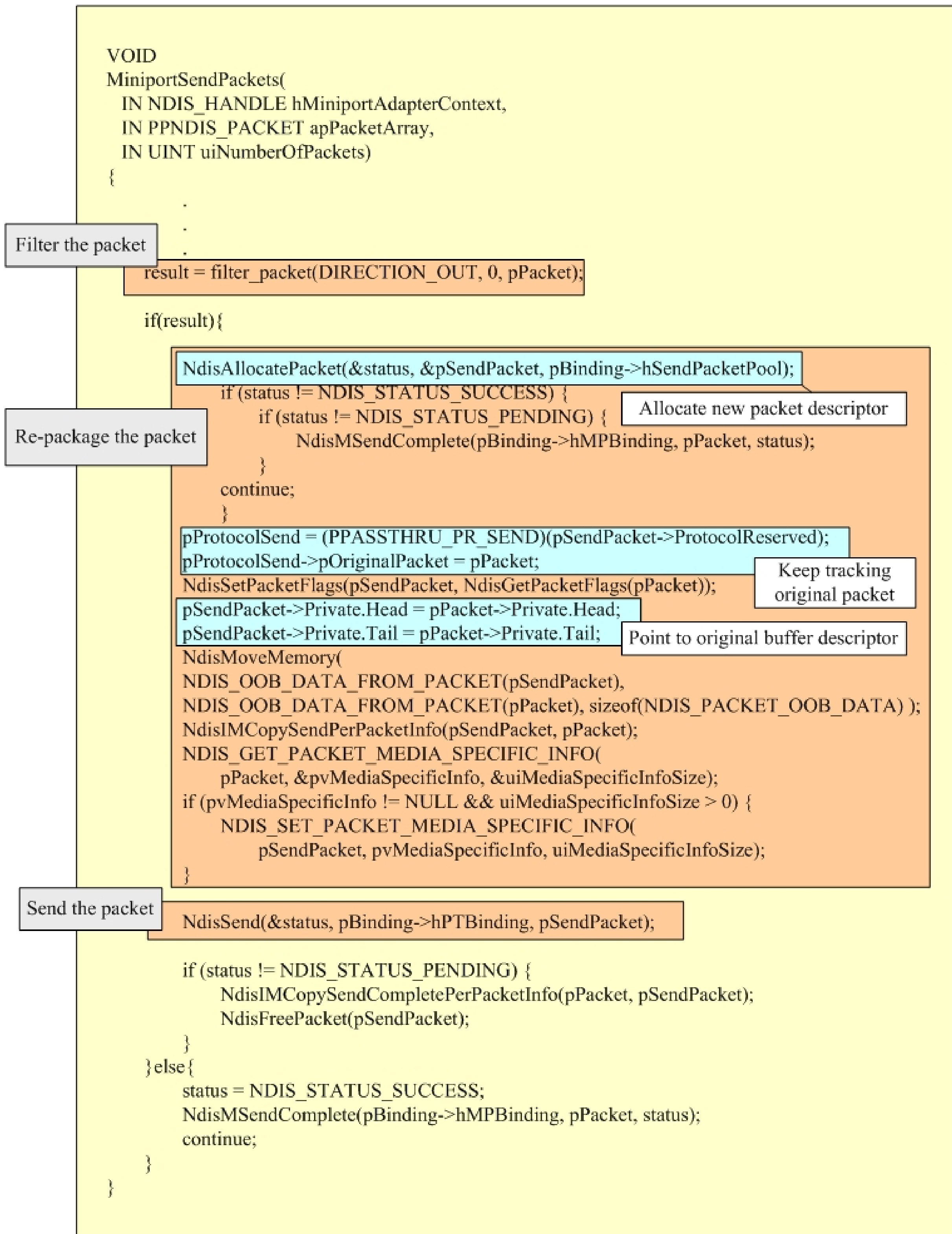



圖 4-20 過濾封包和重新包裝封包的流程範例

### 4.3.3 串流介面驅動程式實作

所謂串流界面驅動程式指的是驅動程式實作了串流界面。而透過實作這個介面可以讓應用程式可以利用檔案輸入輸出的方式來存取驅動程式。一般我們會用表 4-x 中的五個系統函式來存取驅動程式。而要實作串流介面程式，我們必須實作表 4-5 中的 10 個函數。這幾個函式當中，有些是跟表 4-4 有對應關係的，我將再下面一一說明。

```
HANDLE CreateFile(
  LPCTSTR lpFileName, DWORD dwDesiredAccess, WORD dwShareMode,
  LPSECURITY_ATTRIBUTES lpSecurityAttributes, DWORD dwCreationDispostion,
  DWORD dwFlagsAndAttributes,  HANDLE hTemplateFile
);
```

```
BOOL CloseHandle( HANDLE hObject );
```

```
BOOL ReadFile(
  HANDLE hFile, LPVOID lpBuffer, DWORD nNumberOfBytesToRead,
  LPDWORD lpNumberOfBytesRead, LPOVERLAPPED lpOverlapped
);
```

```
BOOL WriteFile(
  HANDLE hFile, LPCVOID lpBuffer, DWORD nNumberOfBytesToWrite,
  LPDWORD lpNumberOfBytesWritten, LPOVERLAPPED lpOverlapped
);
```

```
BOOL DeviceIoControl(
  HANDLE hDevice, DWORD dwIoControlCode, LPVOID lpInBuffer,
  DWORD nInBufferSize, LPVOID lpOutBuffer, DWORD nOutBufferSize,
  LPDWORD lpBytesReturned, LPOVERLAPPED lpOverlapped
);
```

表 4-4 FILE I/O 函式

| |
|---|
| XXX_Init<br>XXX_Deinit<br>XXX_Open<br>XXX_Close<br>XXX_Read<br>XXX_Write<br>XXX_IOControl<br>XXX_PowerUp<br>XXX_PowerDown<br>XXX_Seek |

表 4-5 STREAM INTERFACE FUNCTIONS

下面我將講解在我的驅動程式中，所實作到的串流界面函式：

XXX_Init

```
DWORD XXX_Init(DWORD dwContext);
```

```
DWORD XXX_Init(LPCTSTR pContext, LPCVOID lpvBusContext);
```

當裝置管理員去載入驅動程式的實體時，裝置管理員會去呼叫驅動程式的 Init 函式。然而 Init 函式的宣告型態有上述兩種，前者為舊的型態，但在新版中仍然支援；後者為先的型態。兩種型態的第一個參數，通常傳入的是一個字串指標，指向裝置管理員在 Registry 裡為驅動程式所建立的主動機碼(active key)，所有被裝置管理員所載入的驅動程式都會在 Registry 有此機碼。但是如果驅動程式是被應用程式手動載入，這個參數就不一定是主動機字串的指標。

在 Registry 裡面，存在 HKEY_LOCAL_MACHINE\Drivers\BuiltIn

底下的驅動程式，在系統啟動時都會被裝置管理員自動載入；此外應用程式可以呼叫下面兩個系統函式，來手動載入驅動程式。

```
HANDLE RegisterDevice(
   LPCWSTR lpszType,
   DWORD dwIndex,
   LPCWSTR lpszLib,
   DWORD dwInfo
);
```

```
HANDLE ActivateDeviceEx(
   LPCWSTR lpszDevKey,
   LPCVOID lpRegEnts,
   DWORD cRegEnts,
   LPVOID lpvParam
);
```

如果我們利用 RegisterDevice 來載入我們的驅動程式，Init 的第一個參數就是 RegisterDevice 的第四個參數，所以他可以是任意的數值，端看程式設計師是否需要用這個資料來傳遞特定的資訊。而如果我們呼叫的是 ActivateDeviceEx，Init 的第一個參數跟之前所講的一樣，是驅動程式在 Registry 主動碼字串的指標。而如果 Init 函式所用的是新型態的宣告，則他的第二個參數，就是 ActivateDeviceEx 的第四個參數。

通常我們在這個函式裡面，會去做一些初始化的動作。但由於我的驅動程式並非用來控制實際的硬體裝置。所以在此函式中，我僅去做記憶體的配置和初始化變數和資料結構。

如果載入成功，驅動程式必須回傳 0 以外的值給裝置管理員。這個值通常被稱作裝置本體操作(device context handle)，為一指標用來指向包含驅動程式狀態的資料結構。如果該驅動程式是可多實體(可以被載入超過一次，支援多個硬體裝置實體)，驅動程式本身必須能獨立的維護每一個驅動程式實體本身的狀態，而每次被載入時回傳不一樣的裝置本體操作。在我的實作當中，由於驅動程式只會被我自己手動載入一次，所以回傳的值為一固定常數，並無實質作用。

XXX_Open

```
DWORD XXX_Open(
   DWORD hDeviceContext,
   DWORD AccessCode,
   DWORD ShareMode
);
```

當應用程式呼叫 CreateFile 來開啟驅動程式時 ，這個函式就會被呼叫。他的第一個參數為 Init 函式的回傳值 ，而第二和第三個參數則是對應到 CreateFile 的第二和第三個參數，分別表示存取的形式（read/write 或 read-only）和共享的模式(FILE_SHARE_READ 或 FILE_SHARE_READ)。

當驅動程式接受了應用程式的開啟後，會回傳一個非 0 的數值，這個數值通常用來表示開啟本體(open context)。通常為一個指標，指向一資料結構用來存放開啟的狀態變數。如果驅動程式允許多重存取， 能讓應用程式能同時重複開啟，驅動程式本身就必須能獨立的維護每一個應用程式開啟本體的狀態，而每次被開啟時回傳不一樣的開啟本體。而如果驅動程式本身只被允許開啟一次，則通常我們會直接回傳函式接收到的第一個參數，就是 Init 回傳的裝置本體。在我實作的驅動程式裡，就是這樣的作法。

XXX_DeInit

```
BOOL XXX_Deinit( DWORD hDeviceContext );
```

當驅動程式被卸載（Unload）時，Deinit 函式就會被呼叫。函式的唯一傳入參數為裝置本體，讓 Deinit 函式用來辨別是哪一個驅動程式的實體將被卸載。

在這個函式中，通常是做一些資源的釋放和關閉相對應的硬體。因為我的驅動程式必沒有實質的硬體裝置，所以僅做記憶體的釋放動作。當卸載成功後，則回傳 TRUE 值；如果失敗或有任何錯誤，則回傳 FLASE 值。

XXX_Close

```
BOOL XXX_Close( DWORD hOpenContext );
```

當應用程式呼叫 ClsoeHandle 來關閉它之前所開啟的驅動程式時，Close 函式就會被呼叫。傳入的參數為之前驅動程式的 Open 函式回傳的開啟本體，用來識別關閉的是哪一次的開啟動作。

在這個函式中，通常會釋放用來存放開啟狀態的開啟本體的記憶體空間，但由於我的驅動曾是不支援多重開啟，所以只重設驅動程式狀態到未被開啟的狀態。

XXX_READ

```
DWORD XXX_Read(
   DWORD hOpenContext,
   LPVOID pBuffer,
   DWORD Count
);
```

當應用程式呼叫 ReadFile 系統函式來存取驅動程式時，驅動程式的 Read 函式就會被呼叫。函式的第一個參數為 Open 函式所傳的開啟本體。第二個參數為呼叫應用程式的緩衝區，用來存放要讀取的資料。最後一個參數為緩衝區的大小。如果沒有任何錯誤，將回傳入寫入的緩衝區的位元數。

XXX_WRITE

```
DWORD XXX_Write(
   DWORD hOpenContext,
   LPCVOID pBuffer,
   DWORD Count
);
```

相對於 Read 函式處理讀資料的動作，Write 函式則負責處理讀資料的動作。當應用程式呼叫 WriteFile 系統函式來存取驅動程式時，驅動程式的 Write 函式就會被呼叫。函式的第一個參數為 Open 函式所傳的開啟本體。第二個參數為呼叫應用程式的緩衝區，用來存放要寫入的資料。最後一個參數為緩衝區的大小。如果沒有任何錯誤，將回傳入寫入的位元數。

XXX_IOControl

```
BOOL XXX_IOControl(
    DWORD hOpenContext,
    DWORD dwCode,
    PBYTE pBufIn,
    DWORD dwLenIn,
    PBYTE pBufOut,
    DWORD dwLenOut,
    PDWORD pdwActualOut
);
```

當應用程式呼叫 DeviceIOControl 系統函式來存取驅動程式時，驅動程式的 IOControl 函式就會被呼叫。這是驅動程式和應用程式間一個非常實用而且常用的介面，許多驅動程式常利用這個介面來和應用程式溝通，甚至利用此介面來作讀寫的動作，而不用 Read 和 Write。我實作的非同步輸入輸出動作也是實作在介面上。IOControl 的第一個參數為 Open 函式所傳的開啟本體；而第二個參數為裝置定義的一個控指碼，用來識別控制動作本身。這個控指碼是程式設計師可以自行定義的。我們可以針對不同的控制動作，定義相對應的控制碼。

```
#define MIP_DEVICE_TYPE (32768 + 638)
//
// IOCTLs start here
//
#define MIP_API_BASE (0x800 + 55)

#define CODE_N(n) (n + MIP_API_BASE)
//
// 1. start hooking
//
#define MIP_ENABLE CTL_CODE(MIP_DEVICE_TYPE, CODE_N(0), METHOD_BUFFERED, FILE_ANY_ACCESS)

//
// 2. stop hooking - only the thread that starts a hook can stop it
//
#define MIP_DISABLE CTL_CODE(MIP_DEVICE_TYPE, CODE_N(1), METHOD_BUFFERED, FILE_ANY_ACCESS)

//
// 3. Hook this
//
#define MIP_PUSH_BUFFER CTL_CODE(MIP_DEVICE_TYPE, CODE_N(2), METHOD_OUT_DIRECT, FILE_ANY_ACCESS)

//
// 4. get adapters list
//
#define MIP_ENUM_ADAPTERS CTL_CODE(MIP_DEVICE_TYPE, CODE_N(3), METHOD_OUT_DIRECT, FILE_ANY_ACCESS)

//
// 5. set hook src address
//
#define MIP_SET_IPADDR CTL_CODE(MIP_DEVICE_TYPE, CODE_N(4), METHOD_IN_DIRECT, FILE_ANY_ACCESS)

//
// 6. set hook src address
//
#define MIP_SET_IPIF CTL_CODE(MIP_DEVICE_TYPE, CODE_N(5), METHOD_IN_DIRECT, FILE_ANY_ACCESS)

//
// 7. set block hooked arp packets
//
#define MIP_FLT_ARP CTL_CODE(MIP_DEVICE_TYPE, CODE_N(6), METHOD_IN_DIRECT, FILE_ANY_ACCESS)

//
// 8. set block hooked ip packets
//
#define MIP_FLT_IP CTL_CODE(MIP_DEVICE_TYPE, CODE_N(7), METHOD_IN_DIRECT, FILE_ANY_ACCESS)
```

圖 4-21 定義I/O 控制碼

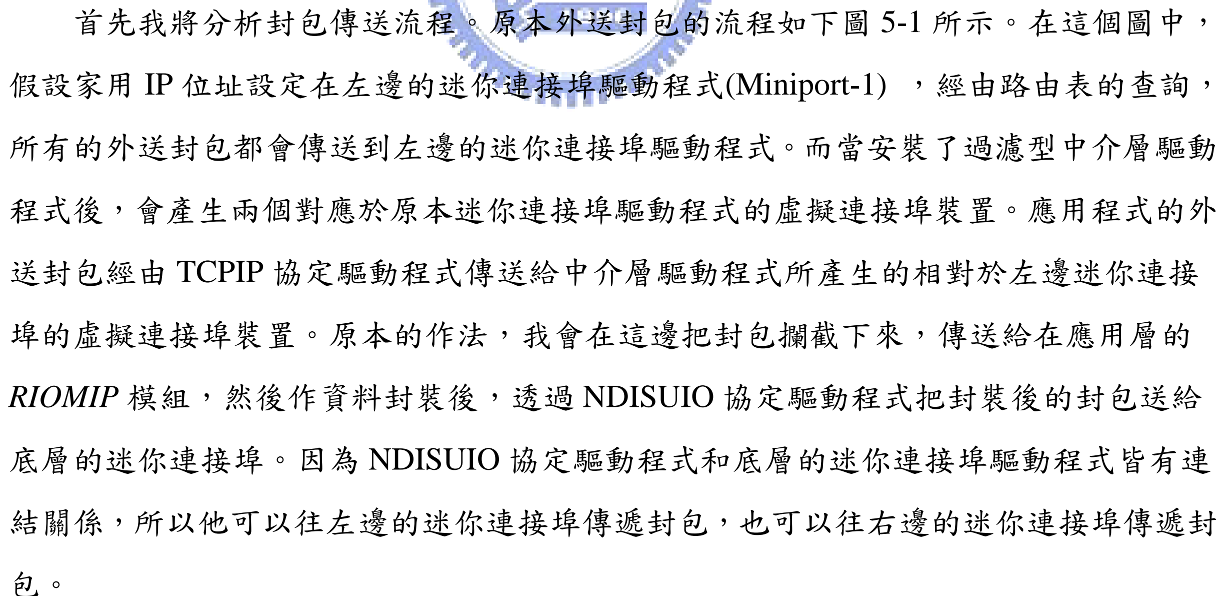
例如，圖 4-21 中為我驅動程式中定義的控制碼。而第三和第四個參數分別是輸入緩衝區的指標和輸入緩衝區大小。輸入緩衝區通常是存放著應用程式要傳遞給驅動程式的資料。第四和第五個參數分別是輸出緩衝區的指標和輸出緩衝區大小。輸出緩衝區通常是永來讓驅動程式寫回資料給應用程式用的。最後一個參數則是最後驅動程式寫道輸出緩衝區的資料位元個數。

# 第五章 RIOMIP 之改進

## 5.1 動機

在 *RIOMIP* 中，應用程式外送的封包都會被攔截和過濾，而要封裝的 IP 封包會經由我們設計的介面，傳送到 *RIOMIP* 在應用層模組進一步封裝，然後在選擇適當的網路介面裝置傳送。很明顯，在封包的傳送過程必須花費額外的時間將封包從驅動程式傳送給在上層的 *RIOMIP* 模組。所以我想進一步的把封裝封包的工作，在底層的網路中介驅動程式完成，直接傳送給迷你連接驅動程式傳送，避免花費額外的時間在將封包傳送給上層的 *RIOMIP* 模組。這樣一來，不僅可以增加網路傳輸的效率，相信也可以減少中央處理器所浪費餓外的計算時間和電力。

## 5.2 問題分析



首先我將分析封包傳送流程。原本外送封包的流程如下圖 5-1 所示。在這個圖中，假設家用 IP 位址設定在左邊的迷你連接埠驅動程式(Miniport-1) ，經由路由表的查詢，所有的外送封包都會傳送到左邊的迷你連接埠驅動程式。而當安裝了過濾型中介層驅動程式後，會產生兩個對應於原本迷你連接埠驅動程式的虛擬連接埠裝置。應用程式的外送封包經由 TCPIP 協定驅動程式傳送給中介層驅動程式所產生的相對於左邊迷你連接埠的虛擬連接埠裝置。原本的作法，我會在這邊把封包攔截下來，傳送給在應用層的 *RIOMIP* 模組，然後作資料封裝後，透過 NDISUIO 協定驅動程式把封裝後的封包送給底層的迷你連接埠。因為 NDISUIO 協定驅動程式和底層的迷你連接埠驅動程式皆有連結關係，所以他可以往左邊的迷你連接埠傳遞封包，也可以往右邊的迷你連接埠傳遞封包。

如果我們想在原來的軟體架構下，在中介驅動成程式作封包的封裝動作，如下圖 5-x 所示。當左邊的虛擬連接埠驅動程式 (virtual adapter-1)接收到封包之後，直接作封包的封裝。做完封裝之後，當要把封裝完的封包往下送時，他只能送給底層左邊的迷你連接埠驅動程式 (Miniport-1)。因為在過濾型中介層驅動程式中，虛擬連接埠裝置和底層實

體裝置的迷你連接埠驅動程式的連結關係是一對一的，左邊虛擬裝置 (virtual adapter-1)只跟左邊的迷你連接埠有連結關係，所以無法傳送封包到右邊的迷你連接埠(如圖 5-2)。如此一來就沒把法完成原本在多網路間切換的功能。

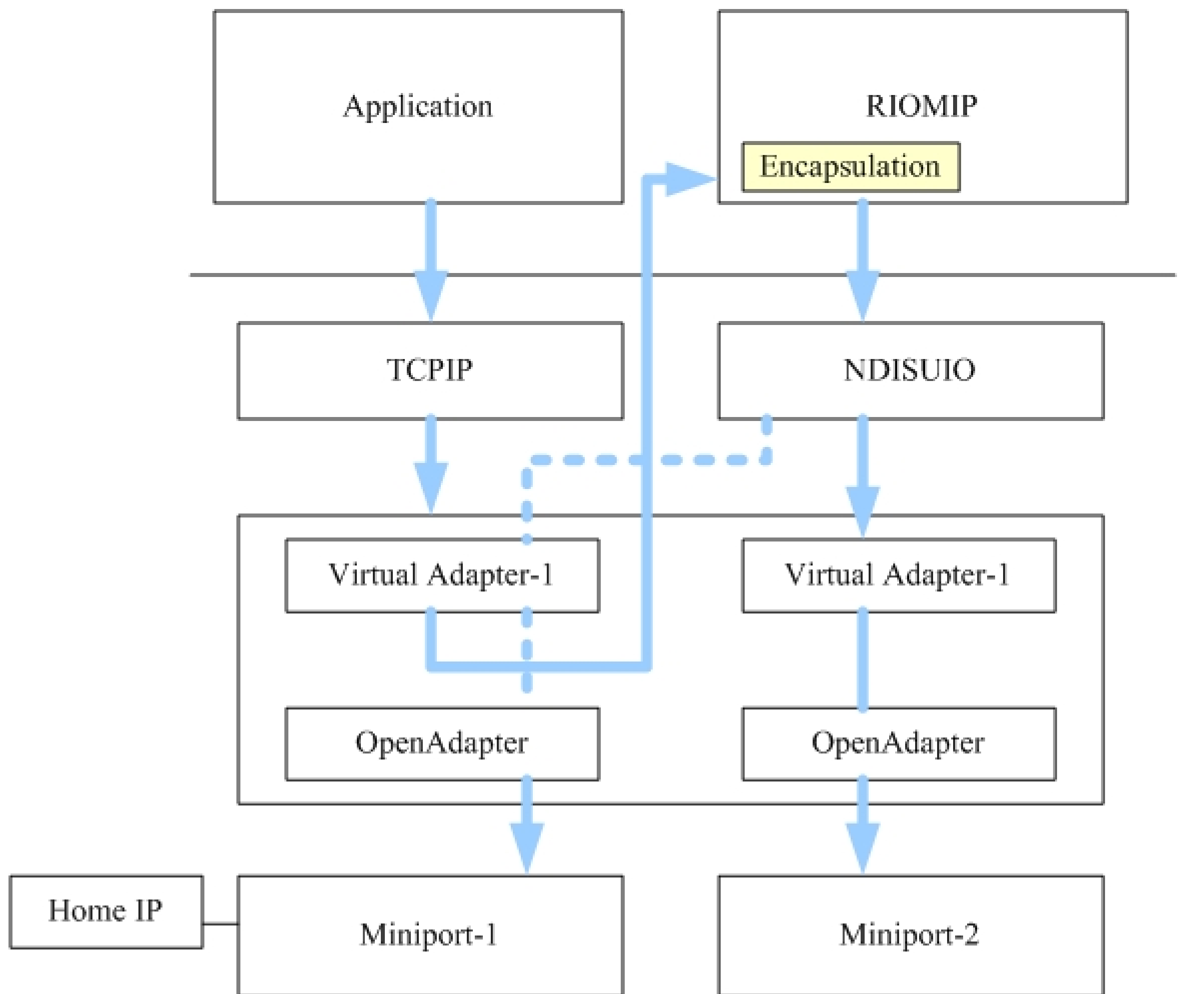



圖 5-1 RIOMIP 外送封包流程

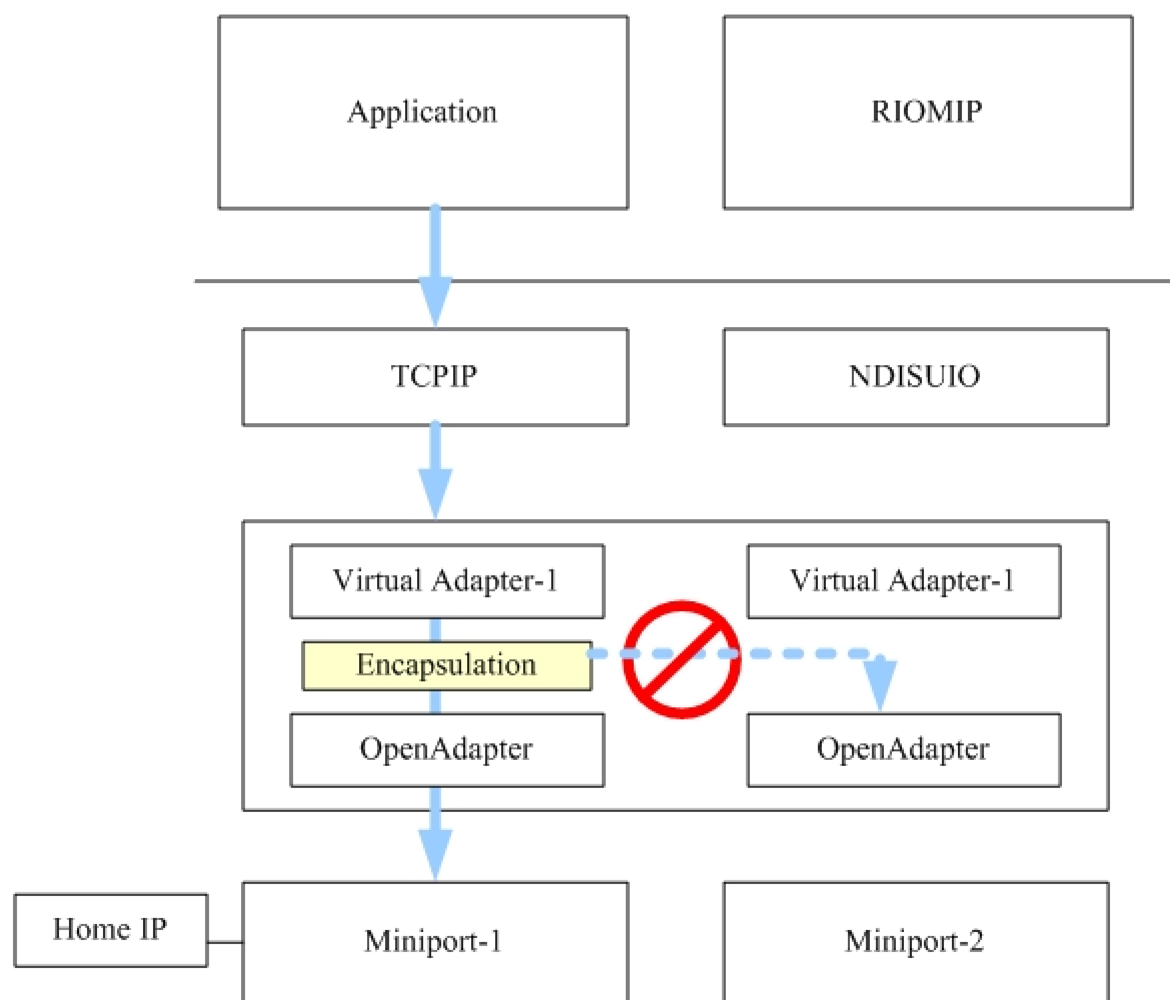



圖 5-2 理想外送封包流程

因為過濾型中介驅動程式先天上的限制，所以如果要想在底層的中介驅動程式作資料的封裝，就得修給原本的過略型中介層驅動程式架構。

## 5.3 解決方案

為了能讓中介層驅動程式可以透過底層所有的迷你連接埠驅動程式傳送封包，所以中介層驅動層式的虛擬網路裝置必須要和所有的實體網路裝置的迷你連接埠驅動程式做連結。所以必須要將原本的過濾型中介層驅動程式的實作方式改為多路傳輸中介層驅動程式的實作方式。

為了達到在中介層驅動程式封裝封包，並且達到能夠在多個網路裝置間切換的功能，所以我將原本的過濾型驅動程式，改為一對多型的多路傳輸中介驅動程式。所謂一對多意指虛擬連接埠裝置和底層實體裝置的迷你連接埠驅動程式的連結關係是一對

多。中介層的驅動程式只會產生一個虛擬連接埠裝置，並自己維護和底層真實網路裝置的迷你連結埠驅動程式的聯結關係。整個新的多路傳輸中介驅動程式及外送封包流程如圖 5-3 所示。

多路傳輸中介層驅動程式會產生一個虛擬網路裝置用來設定家用 IP 位址，所有應用程式外送的封包都會經由 TCPIP 協定驅動程式傳送給虛擬網路裝置。虛擬網路裝置在收到封包後，進行封包的封裝。因為虛擬連接埠和維護了本身和底層實體迷你連接埠之間的連結，所以它可以選擇底層的實體迷你連接埠驅動程式，將封包傳送出去。

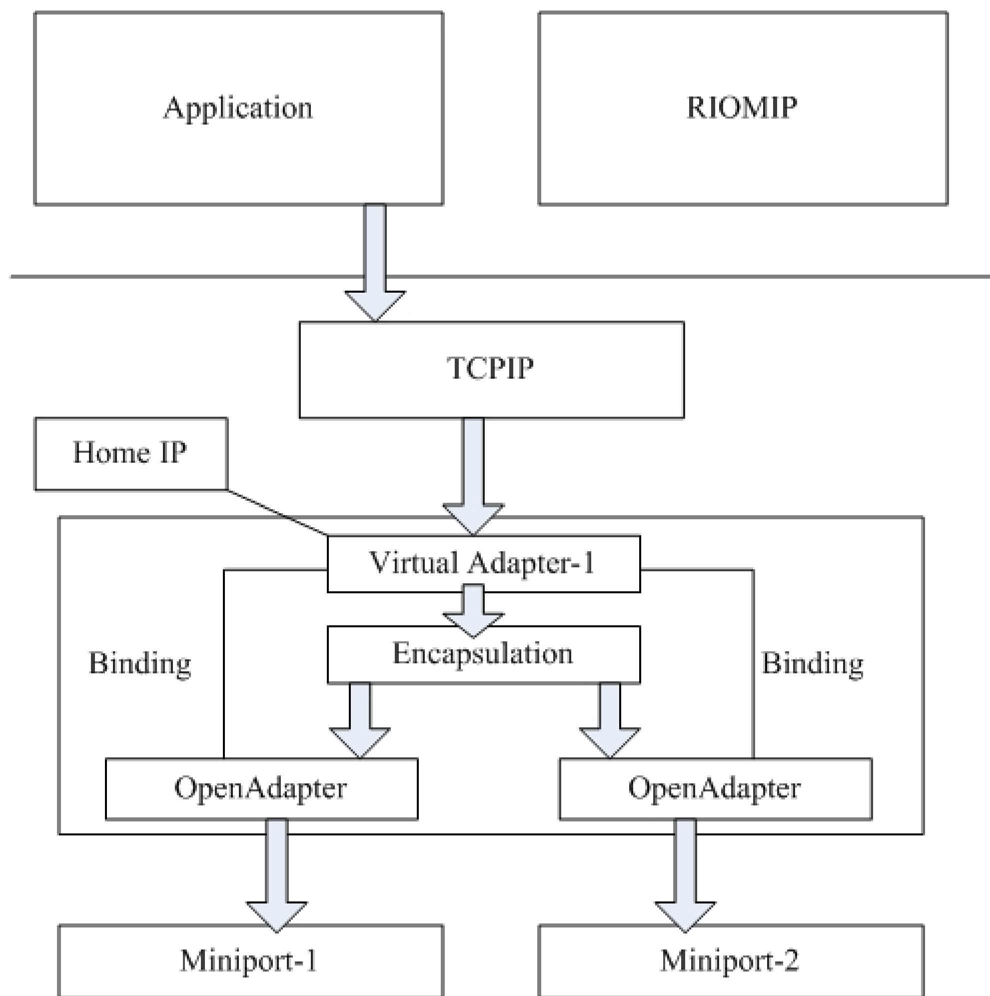



圖 5-3 新的多路傳輸中介驅動程式及外送封包流程

進一步，當做封包封裝的時候，會需要一些額資訊，是虛擬連接裝置所無法得知的。主要包括 Mobile IP 的狀態和封裝所需要的網路層資訊。虛擬連接裝置在做封裝之前，必須先取得現在所處的網域，來決定是否需要做封裝的動作。如果在家用網域裡面，就不需要做封裝的動作。另外，也必須知道所要使用的封裝方式，這也是在註冊的時候所

決定的。而進行封裝的時候，會需要到一些網路層的資訊，包括底層實體網路介面所使用的轉交位址 (care-of-address) ，和他所聯結的閘道器的實體位址。因為虛擬連接埠處在 TCPIP 協定驅動程式的下方，所以並沒有直接的方法可以取的這些資訊。

關於這些資訊的缺乏，必須仰賴在應用層的 *RIOMIP* 模組，透過中介層驅動程式串列介面的實作，利用 DeviceIOControl 系統函式，將這些資訊傳遞給中介層驅動程式。

## 5.4 實作細節

### 5.4.1. 多路傳輸中介層驅動程式實作

實作多路傳輸中介層驅動程式和過濾型中介層驅動程式一樣，必須實作協定層驅動程式介面和迷你連接埠驅動程式兩個介面。差別在於虛擬連接埠和底層實體連接埠驅動程式之間的連接關係。為了實現一對多的連接關係，重點在於協動驅動程式介面的 ProtocolBindAdapter 函式。

- ProtocolBindAdapter：當每一個網路介面卡被迷你連接埠初始化後，NDIS 都會主動來呼叫這個函式，來要求協定驅動程式去跟底層的迷你連接埠驅動程式做連結，並取得底層迷你連接埠驅動程式的操作物件，用以存取底層的迷你連接埠驅動程式，例如傳送封包等。NDIS 函式庫回傳入底層迷你連接埠的裝置名稱，而藉由呼叫 NdisOpenAdapter 函式我可以去連結底層的網路界面裝置並取得連接埠驅動程式的操作物件。NdisOpenAdapter 的定義如下所示，第三個參數即是所傳回的即是底層迷你連接埠驅動程式的操作物件

```
VOID NdisOpenAdapter(
  PNDIS_STATUS Status,
  PNDIS_STATUS OpenErrorStatus,
  PNDIS_HANDLE NdisBindingHandle,
  PUINT SelectedMediumIndex,
  PNDIS_MEDIUM MediumArray,
  UINT MediumArraySize,
  NDIS_HANDLE NdisProtocolHandle,
  NDIS_HANDLE ProtocolBindingContext,
  PNDIS_STRING AdapterName,
  UINT OpenOptions,
  PSTRING AddressingInformation
);
```

因為要實做中介層驅動程式，在這個函式裡必須去產生虛擬連接埠裝置。而透過呼叫 NdisIMInitializeDeviceInstanceEx 函式，我們可以產生一個虛擬連結埠。而之前過濾型的中介層驅動程式中，虛擬連接埠裝置根底層的實體網路介面卡的對應

關係是一比一的，所以對於每個實體的網路介面卡，都必須產生一個虛擬連接埠裝置。也就是這個函式每被呼叫一次，就必須產生一個虛擬連接裝置。而在這邊，我們因為要實作的是一對多的多路傳輸中介層驅動程式，所以我們只有再第一次被呼叫的時候會呼叫 NdisIMInitializeDeviceInstanceEx 函式產生唯一的虛擬連接埠裝置。而之後被呼叫時，只會呼叫 NdisOpenAdapter 函式開啟與下面迷你連接埠的連結，並紀錄該迷你連接埠的操作物件。為了維護這樣一個內部的連結關係，我們定義如下的資料結構去紀錄連結的關係：

```
typedef struct    _BundleAdapter{
        BOOLEAN              deviceCreated;
        NDIS_HANDLE          hMPBinding;
        NDIS_STATUS          statusLastIndicated;
        BINDING*             bundleAdapters[MAX_BUNDLE_ADAPTERS];
        int                  numBundleAdapter;
        int                  primaryAdapter;
}BundleAdapter, *PBundleAdapter;
```

- deviceCreated: 用來標記虛擬連接埠裝置是否建立，如果否則會呼叫 NdisIMInitializeDeviceInstanceEx 函式產生唯一的虛擬連接埠裝置。
- hMPBinding: 用來紀錄中介層驅動程式的迷你連接埠端和上層協定驅動程式的連接關係。為當要與上層的協定驅動程式溝通時，所需要的操作物件。
- StatusLastIndicated: 紀錄上次底層迷你連接埠驅動程式所告知到狀態。
- bundleAdapters: 用來紀錄與底層迷你連接埠的關係。存放所有底層迷你連接埠的操作物件，依序編號存在陣列。
- numBundleAdapter: 底層有建立連結的迷你連接埠各數
- primaryAdapter: 現在所使用的底層實體連接埠的編號

### 6.4.2 封包的操作和封裝

在前面的章節我介紹過 NDIS 描述封的方式，不過因為實作之前的過濾型中介層驅動程式時，並不需要對封包作修改的動作，所以在這邊我將詳細的描述怎麼去修改封包的內容，作封裝的動作。底下我將分步驟說明封包操作和進行封裝的過程：

1. 取得封包的內容: 這部分我在之前的章節有提到過，我在這邊在簡短的描述一次。因為在對於上層協定驅動程式所傳送果過來的封包，我們所取得的是一個封包描述子，用 NDIS_PACKET 資料結構來表示。利用 NdisQueryPacket 函式，可以取

得串接在該封包描述子的緩衝區的緩衝區描述子。在利用 NdisQueryBuffer 來取的緩衝區描述子所描述的緩衝區記憶體地址。透這記憶體地址指標，我們就可以存取封包的內容。如果串接在該封包描述子的緩衝區不止一個，可以繼續呼叫 NdisGetNextBuffer 來取得下一個緩衝區的描述子。

2. 修改封包內容: 由於步驟 1 取得包含封包內容的緩衝區屬於上層的協定驅動程式。對中介層驅動程式來說，只能把它當作唯讀的。所以我們要修改封包之前，必須先配置屬於中介層驅動程式的緩衝區，再把封包的內容拷貝到新的緩衝區。然後根據封裝的模式，加入適當檔頭。

3. 包裝成新的封包描述子: 因為要將封包傳送出去，我們必須重新配置一個新的封包描述子。在這邊先呼叫 NdisAllocatePacket 函式來配置一個新的封包描述子。再者，也需要配置新的緩衝區描述子來描述儲存封裝完封包的緩衝區。藉由呼叫 NdisAllocateBuffer 函式來配置新的緩衝區描述子。最後藉由呼叫 NdisChainBufferAtFront 把緩衝區串接到新的封包描述子。

# 第六章 跨階層網路事件傳遞中介軟體設計與實作

本章介紹我在 Windows CE 作業平台實作的跨階層網路事件傳遞中介軟體。這個中介軟體作主要能讓上層的應用程式知道底層網路的狀況。應用程式可利用中介軟體的函式庫，訂閱一些特定的網路事件；另外，中介軟體也提供底層網路的資訊和參數，讓應用程式可以查詢和設定。

## 6.1 跨階層網路事件傳遞中介軟體概觀

跨階層網路事件傳遞中介軟體的設計概念是基於網路跨階層設計的概念，目的是讓程式設計師更容易開發出在行動端點上的具有高效率行動性的軟體。而當行動端點上具有多個異質網路通訊裝置時，跨階層網路事件傳遞中介軟體也能幫助程式設計師開發出能在異質網路間漫遊的應用程式。

而本中介軟體最主要的功能，就是透過中介軟體的應用程式開發介面，程式設計師可以很容易取得底層網路的資訊。為了達到這樣的功能，在我們的中介軟體中定義了兩種型態的變數，分別來描述網路的狀況:

- 事件 (event) 變數：特定的網路事件，例如行動端點網路位址的改變，網路媒介連線中斷等。當網路事件發生時，中介軟體能透過特定的機制來告知應用程式，而程式設計師可以針對特定網路事件，在應用程式中作特殊的處理。
- 狀態 (state) 變數：為一些參數，能夠代表現在網路的狀態和相關資訊。狀態變數是的操作方式有兩種，設定和查詢。但並非所有的參數都可以設定，有的狀態變數只能支援查詢的動作。

底下表 6-1 分別列出跨階層網路事件傳遞中介軟體所支援的事件和狀態變數。 L2 代表網路第二層，即連結層 (Link Layer)；L3 代表網路的第三層，即網路層 (Network Layer)。

| Layer | Event | State Variable |
|---|---|---|
| L2 | LINK_UP<br>LINK_DOWN<br>DOT11_RSSI_TRIGGER | MEDIA_IN_USE<br>MEDIA_CONNECT_STATUS<br>DOT11_BSSID<br>DOT111_BSSID_LIST<br>DOT11_SSID<br>DOT11_BSSID_SCAN<br>DOT11_POWER_LEVEL |
| L3 | IP_CHANGE<br>GATEWAY_CHANGE<br>ROUTING_CHANGE<br>MIP_IN_HOME<br>MIP_AWAY_HOME | MIP_ENABLE<br>MIP_MODE<br>.<br>.<br>. |

表 6-1 跨階層網路事件傳遞中介軟體事件和狀態變數

## 6.2 驅動程式和應用程式間訊系的傳遞

在實作這個中介軟體時，有個很重要的問題，就是當底層的驅動程式如何主動去通知上層的應用程式特定事件的發生。通常，應用程式可以藉由檔案輸入輸出的介面，利用 DeviceIOControl 的系統函式，將特定的事件或資料傳遞給驅動程式。但 Windows 系列作業系統中，並無特定的機制來讓驅動程式主動和應用程式溝通。底下我將針對幾個可行的方法作討論。

### 6.2.1 共享事件

關於這個方法，在前面討論實作非同步輸入輸出時就已經有討論過。基本上是利用 Windows 提供的事件 (event) 物件來實作。應用程式先建立一個事件物件，利用 DeviceIOCotrol 系統函式主動傳遞給驅動程式。驅動程式把這個物件的參考指標儲存下來，等到需要告知應用程式特定事件時，再呼叫系統函式去觸發該事件。這個方式，在 Windows NT 系列平台也可以實作。

在 Windows NT 系列平台中，是一個多重記憶體位址的架構，每個程序有自己的記憶體空間。而由於驅動程式所使用的是系統記憶體空間，所以必須需把應用程式傳送過來的事件物件的參考指標對應到系統記憶空間。另外我們必須確保驅動程式是處在該應用程式的本體 (context) 裡面，因為這樣應用程式這樣傳遞過來的事件物件參考指標的

虛擬記憶體位址才會是正確的被對應到該事件物件上。這個條件是限制了該驅動程式必須是直接被該應用程式呼叫，中間不能有其他層疊 (layered) 的驅動程式。

### 6.2.2 轉化呼叫模型 (Inverted Call Model)

這個方法可以讓驅動程式來呼叫使用者空間的函式，也能讓驅動程式傳送資料給應用程式，同時也能讓驅動程式傳遞訊息給應用程式。它的原理是利用前面所提到過的 IRP 資料結構的特性。當驅動程式接收到應用程式所發出的輸入輸出請求時，裝置管理員會配置一個相對應的 IRP 資料結構傳送給驅動程式。驅動程式可以把這個 IRP 標記為待處理 (pending)，必把它存放到佇列當中。我用底下的流程來解說這個方法的運作原理：

1. 應用程式發出一個同步或非同步的輸入輸出請求(呼叫 DeviceIoControl)給驅動程式。這邊指的同步或非同步指的是應用程式發出請求的執行緒是否要等待這個輸入輸出動作的完成。如果是同步輸入輸出，發出該請求的執行緒會被輸入輸出請求系統函式所阻擋 (blocking)，直到輸入輸出動完成，該系統函式才會返回；如果是非同步輸入輸出，應用程式可以利用 WaitForSingleObject 等系統函式去等待該輸入輸出動作的完成。
2. 驅動程式把 IRP 標記為代處理，並把它放到佇列裡面。
3. 當事件發生，驅動程式會去結束這個 IRP。
4. 如果應用程式是使用同步輸入輸出請求，這會讓應用程式發出輸入輸出請求的執行緒從該系統函式 (如 DeviceIoControl) 中返回；如果應用程式是使用非同步輸入輸出請求，它觸發法非同步輸入輸出事件的發生。

這個方式的好處在於它不受呼叫本體的限制，驅動程式不必執行在該應用程式的本體裡。不過由於 Windows CE 作業平台並不支援非同步輸入輸出，亦沒有 IRP，所以不適用此方法。

### 6.2.3 命名事件 (Named Event )

在 Windows 系列系統中，事件物件是可以被命名的。我們在建立事件物件時可以同時賦予它一個名稱，這個事件就被稱為命名事件 (Named Event)。我們可以在驅動程式

裡呼叫 IoCreateNotificationEvent 系統函式建立一個命名事件，然後讓應用程式去開啟並等待該事件。當特定事件發生後，驅動程式就可以去觸發該命名事件來告知應用程式。不過這個方法有個很大的問題，如果該事件是在驅動程式的進入點函式 (DriverEntry) 中被建立，因為它是該函式是在系統程序的本體中執行，所以它會繼承系統程序的安全屬性。所以，如果應用程式不是用系統管理者的權限執行時，開啟該命名物件將會發生錯誤。另外，命名事件也有命名空間衝突的問題，而微軟官方也不建議使用這個發法。

表 6-2 列出了各個方式的優缺點。我選擇了利用共享事件的方式來實作驅動程式和應用程式間事件的傳遞。因為它在 Windows 系列平台都非常容易的實現，所以將來把本中介軟體移植到 Windows XP 時也會容易很多。雖然共享事件有執行本體的限制，但由於驅動程式也是有我自己設計與實作，所以實質上可以輕易避免這個問題。

| | Shared Event | Pending IRP | Named Event |
|---|---|---|---|
| Advantage | Well supported For All Windows OS | - safe for any context | - safe for any context |
| Drawback | - driver must run on application calling context | - not supported in Windows CE | - collision of namespace<br>- Security attribute problem<br>- deprecated by MS |
| Windows XP | O | O | O |
| Windows CE | O | X | O |

表 6-2 驅動程式和應用程式間訊系的傳遞方式比較

## 6.3 跨階層網路事件傳遞中介軟體系統架構

我的跨階層網路事件傳遞中介軟體最主要包括了兩個部份：一個是在使用者層級讓應用程式呼叫的動態連結函式庫；一個是在核心層級負責觀察網路事件的驅動程式。

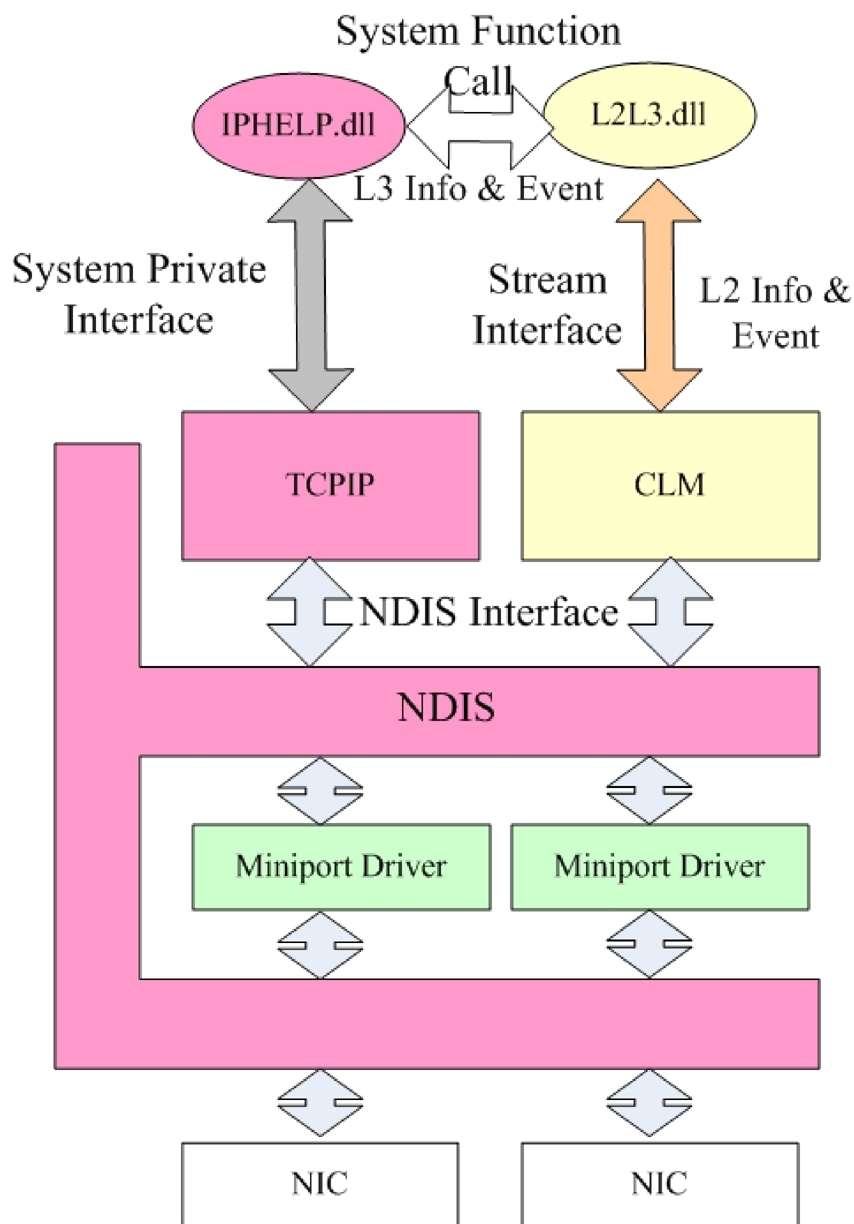


圖 6-1 跨階層網路事件傳遞中介軟體系統架構

### 6.3.1 L2L3 動態函式庫

L2L3 動態函式庫是提供一套應用程式開發介面，讓程式設計師能夠使用跨階層網路事件傳遞中介軟體來訂閱網路事件，設定和查詢網路狀態變數。L2L3 動態函式庫利用系統所提供的 IPHelpr 動態函式庫來向 TCP/IP 協定驅動程式取得第三層，即網路層的網路資訊，同時註冊網路位址改變和路由表改變的事件。另一方面藉由串流界面驅動程式的實作，L2L3 動態函式庫透過檔案輸入輸出的介面， 利用系統函式 DeviceIOControl 來和 CLM 協定驅動程式溝通，取得網路資訊和訂閱網路事件。

在 L2L3 動態函式庫中，提供了表 6-3 中的函式給程式設計使用。

| | |
|---|---|
| Open_CLM | 初始化跨階層網路事件傳遞中介軟體 |
| CLM_Subscribe_Event | 訂閱特定的網路事件 |
| CLM_Emun_Adapter | 查詢中介軟體可用的網路界面裝置 |
| CLM_Query_State | 設定網路狀態變數 |
| CLM_Set_State | 查詢網路狀態變數 |
| Close_CLM() | 停止使用網路事件傳遞中介軟體 |

表 6-3 L2L3 動態函式庫函式表

### 6.3.2 CLM 協定驅動程式

CLM 驅動程式是我實作的一個 NDIS 協定驅動程式，他的作用主要是透過 NDIS 的介面來向下層的迷你連接埠驅動程式查詢和設定網路參數，並觀察特定的網路事件。另一方面，CLM 驅動程式也是個串流界面驅動程，因為他實作了串流介面，讓在使用者層級的 L2L3 動態函式庫可以跟他溝通。

## 6.4 跨階層網路事件傳遞中介軟體運作原理

### 6.4.1 中介軟體實初始化動作

當要使用跨階層網路事件傳遞中介軟體時，必須先呼叫 Open_CLM 函式 來作初始化的動作。該函式最重要的動作是開啟 CLM 協定驅動程式。透過呼叫 CreateFile 系統函式去開啟 CLM 協定驅動程式，取得該驅動程式的操作物件，之後才可以利用 DeviceIOControl 系統函式來存取驅動程式。

### 6.4.2 查詢網路裝置

因為大部分的網路事件始針對某個特定的網路裝置，所以在訂閱網路事件時，必須先知道要有哪些網路裝置。藉由呼叫 CLM_Emun_Adapter 我們可以得知跨階層網路事件傳遞中介軟體所以可以操作的網路裝置。CLM_Emun_Adapter 會利用 IpHelper 函式庫的 GetAdaptersInfo 系統函式去取得系統中所有的網路裝置列表；另一方面，透過 DeviceIOControl 向 CLM 協定驅動程式取得 CLM 協定驅動程式所可以存取的網路裝置，在將兩者做比對的動作，編排出統一的指標值，方便之后的存取。

### 6.4.3 訂閱網路事件

當呼叫 CLM_Subscribe_Event 來訂閱網路事件時，首先函式庫會幫應用程式建立一個事件物件。然後根據所訂閱的事件層級的不同而有不同的動作：

- L3 網路層事件: 如果所訂閱的事件為網路層的事件，在這邊作主要會呼叫 IpHelper 相關的系統函式。最主要為 NotifyAddrChange 和 NotifyRouteChange。NotifyAddrChange 允許我們傳入一個事件的指標，當某個介面所對應的 IP 網路位址改變時，系統就會去觸發該事件。NotifyRouteChange 函式的用法和 NotifyAddrChange 一樣，它則用來觀察系統的路由表是否有所改變。底下做不同事件所作的處理做個別的描述：
  - IP_CHANGE: 呼叫 NotifyAddrChange，來觀察是否有某個網路介面裝置所對應的 IP 網路位址改變。當所傳入的事件被觸發時，會進一步檢查改變的介面裝置是否和訂閱事件時所指定的網路介面裝置一樣，如果是，則進一步去觸發訂閱事件時所傳入的事件來通知應用程式；如果否，重設函式庫本身所建立的事件，繼續呼叫 NotifyAddrChang。如果沒有指定觀察的裝置，則省略檢查的動作。
  - GATEWAY_CHANGE: 呼叫 NotifyRouteChange 函式，還觀察系統的路由表是否有所改變，當路由表改變時，檢查是否為指定的網路介面裝置的閘道器的網路位止改變。如果是觸發發該事件，如果否再次呼叫 NotifyRouteChange 觀察。
  - ROUTE_CHANGE: 呼叫 NotifyRouteChange 函式還觀察系統的路由表是否有

所改變，當路由表改變時，則觸發該事件。

- L2 連結層事件: 如果所訂閱的事件為連結層的事件，函式庫會透過 DeviceIoControl 來向 CLM 協定驅動程式訂閱該事件。函式庫會將訂閱的事件名稱和相對應的事件物件指標傳遞給 CLM 協定驅動程式，當驅動程式觀察到事件的發生時，就會觸發該事件。

NDIS 函式庫提供了一個函式，NdisMIndicateStatus 來讓迷你連接埠驅動程式告知上層的協定驅動程式某些特定的網路事件。例如最常見的就是網路連線的中斷和連接。這邊的網路連線指的是連階層的連線，例如網路線的插入與拔除和無線區域網路是否連上存取點。在 NDIS 裡面分別定義了下面兩個狀態來代表這兩個事件:

- NDIS_STATUS_MEDIA_DISCONNECT
- NDIS_STATUS_MEDIA_CONNECT

而在協定驅動程式，我們可以實作並註冊 ProtcolStatus 處理函式，當底層的迷你連接埠驅動程式呼叫 NdisMIndicateStatus 時，NDIS 函式庫就會來呼叫協定驅動程式的 ProtcolStatus。所已經由實作這個函式，CLM 協定驅動程式就可以觀察一些網路事件的發生。

整個 L2 網路事件訂閱的流程如下圖 6-2 所示

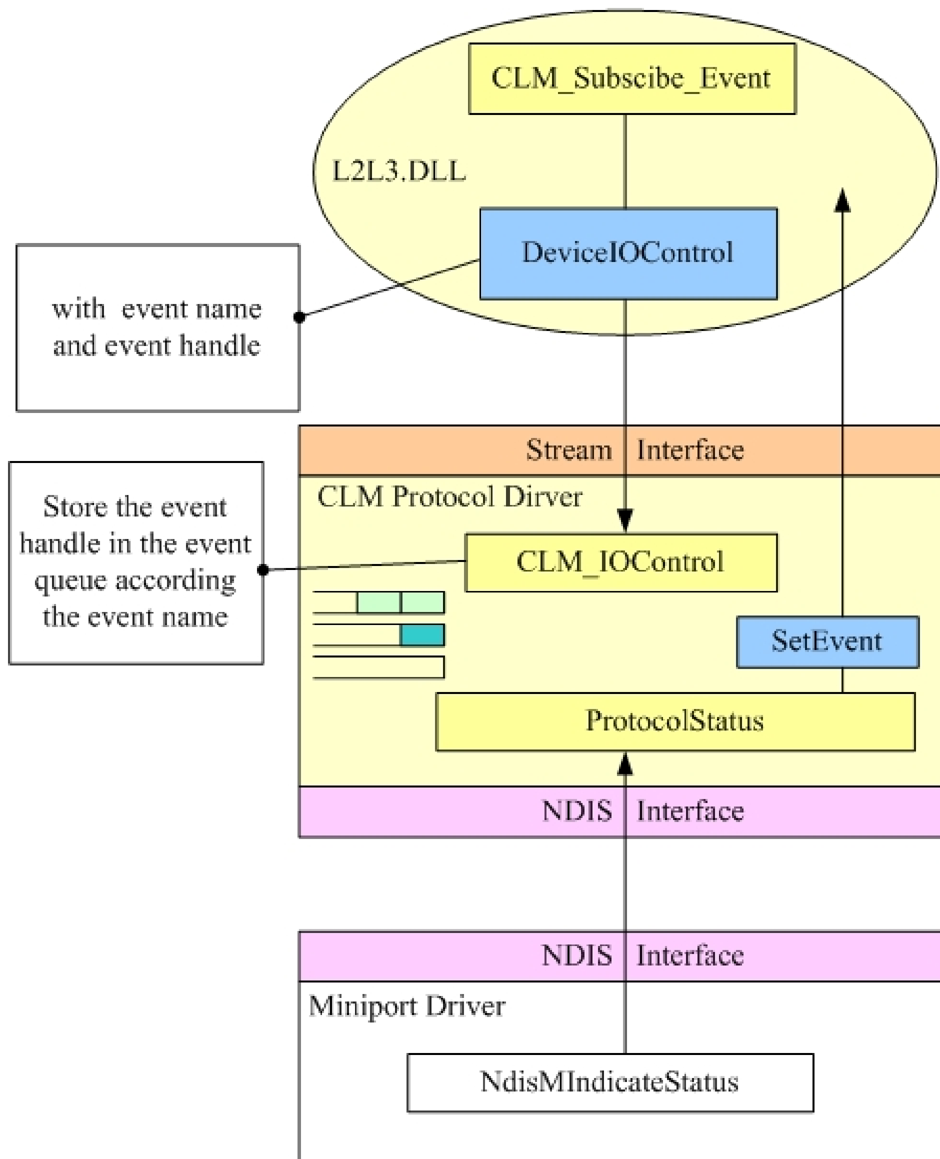



圖 6-2 跨階層網路事件傳遞中介軟體系統架構

### 6.4.4 設定和查詢網路狀態變數

當呼叫 CLM_Query_State 去查詢網路的狀態變數時，函式庫會去呼叫 DeviceIoControl 把要查詢的網路介面裝置名稱，狀態變數名稱和儲存狀態變數的的緩衝區指標傳遞給 CLM 驅動程式。驅動程式會根據所要查詢的狀態變數名稱轉換成相對應的 NDIS 查詢請求，向底層的網路介面裝置的迷你連接埠驅動程式查詢。整個設定和查詢狀態變數運作的流程如下圖 6-3 所示。

在 NDIS 裡面，定義了許多 NDIS object identifiers (OIDs)，用來代表許多系統定義的網路參數， 例如表 6-4 為一般網路型態所共用的 OIDs。另外 NDIS 還有許多針對特

殊網路型態所地定義的 OIDs，如乙太網路介面，無線區域網路介面和廣域網路介面等。NDIS 協定驅動程式透過 NdisRequest 函式，可以向底層的迷你連接埠驅動程式發出查詢或設定 OID 的請求。CLM 協定驅動程式就是藉由這樣的方式來取得相關的網路資訊和設定相關的網路參數。

| Name | Description |
| --- | --- |
| OID_GEN_SUPPORTED_LIST | List of supported OIDs |
| OID_GEN_HARDWARE_STATUS | Hardware status |
| OID_GEN_MEDIA_SUPPORTED | Media types supported (encoded) |
| OID_GEN_MEDIA_IN_USE | Media types in use (encoded) |
| OID_GEN_MAXIMUM_LOOKAHEAD | Maximum in bytes, receive lookahead size |
| OID_GEN_MAXIMUM_FRAME_SIZE | Maximum in bytes, frame size |
| OID_GEN_LINK_SPEED | Link speed in units of 100 bps |
| OID_GEN_TRANSMIT_BUFFER_SPACE | Transmit buffer space |
| OID_GEN_RECEIVE_BUFFER_SPACE | Receive buffer space |
| OID_GEN_TRANSMIT_BLOCK_SIZE | Minimum amount of storage, in bytes, that a single packet occupies in the transmit buffer space of the NIC |
| OID_GEN_RECEIVE_BLOCK_SIZE | Amount of storage, in bytes, that a single packet occupies in the receive buffer space of the NIC |
| OID_GEN_VENDOR_ID | Vendor NIC code |
| OID_GEN_VENDOR_DESCRIPTION | Vendor network card description |
| OID_GEN_VENDOR_DRIVER_VERSION | Vendor-assigned version number of the driver |
| OID_GEN_CURRENT_PACKET_FILTER | Current packet filter (encoded) |
| OID_GEN_CURRENT_LOOKAHEAD | Current lookahead size in bytes |
| OID_GEN_DRIVER_VERSION | NDIS version number used by the driver |
| OID_GEN_MAXIMUM_TOTAL_SIZE | Maximum total packet length in bytes |
| OID_GEN_PROTOCOL_OPTIONS | Optional protocol flags (encoded) |
| OID_GEN_MAC_OPTIONS | Optional NIC flags (encoded) |
| OID_GEN_MEDIA_CONNECT_STATUS | Whether the NIC is connected to the network |
| OID_GEN_MAXIMUM_SEND_PACKETS | The maximum number of send packets the driver can accept per call to its *MiniportSendPackets* function |

表 6-4 一般網路型態所共用的 OIDs

相關狀態變數和相對應的 OIDs 如下表 6-5 所示。

| OIDs Name | Event Name | Description |
|---|---|---|
| OID_GEN_MEDIA_IN_USE | MEDIA_IN_USE | Media types in use |
| OID_GEN_MEDIA_CONNECT_STATUS | MEDIA_CONNECT_STATUS | Whether the NIC is connected to the network |
| OID_802_11_BSSID | DOT11_BSSID | the MAC address of the associated access point |
| OID_GEN_MEDIA_IN_USE | DOT111_BSSID_LIST | a list containing all BSSIDs and their attributes |
| OID_802_11_SSID | DOT11_SSID | the SSID with which its NIC is associated |
| OID_802_11_BSSID_LIST | DOT11_BSSID_SCAN | requests the miniport driver to direct the 802.11 NIC to request a survey of BSSs |
| OID_802_11_POWER_MODE | DOT11_POWER_LEVEL | power mode of the 802.11 NIC |

表 6-5 OIDs 和事件對應

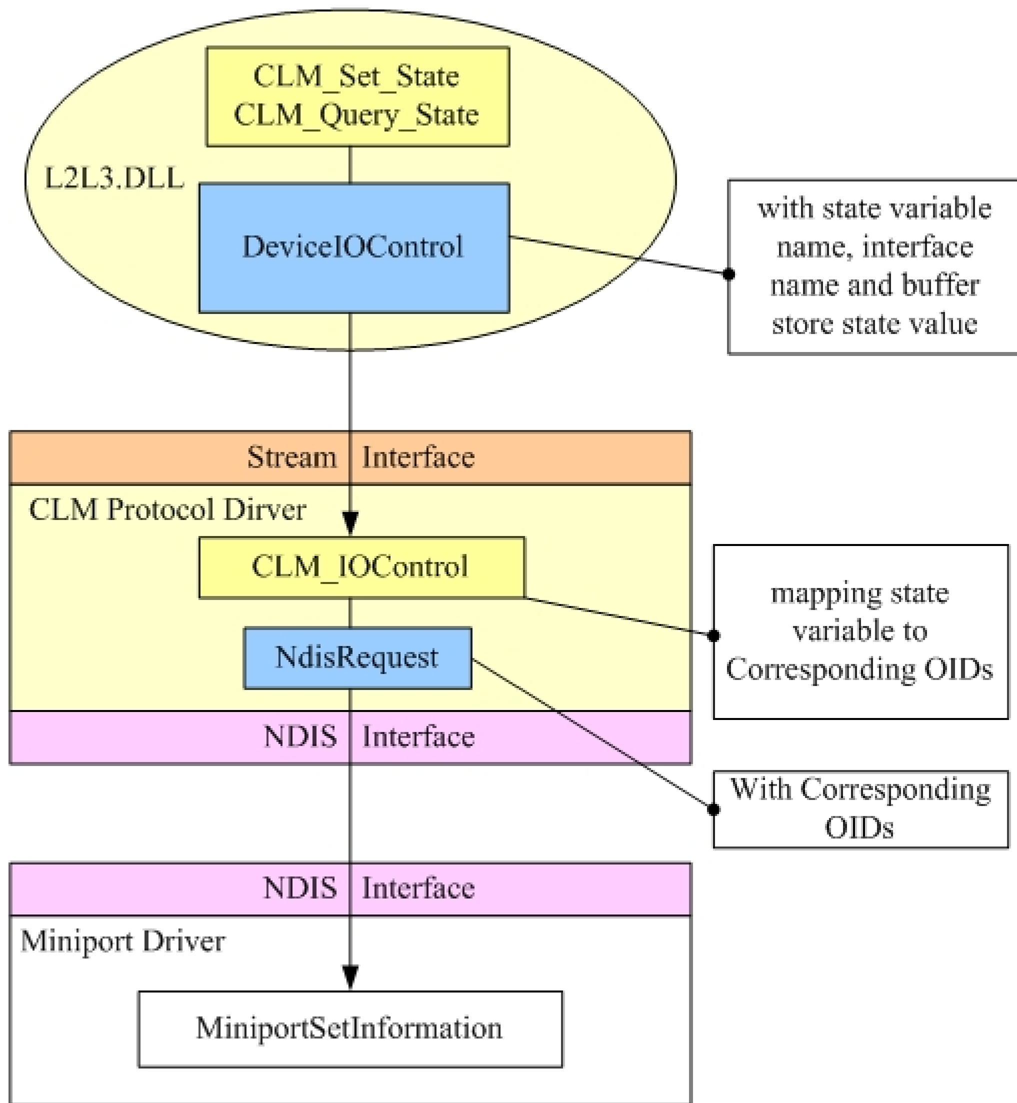



圖 6-3 設定和查詢狀態變數運作的流程

## 6.5 CLM 協定驅動程式的實作

CLM 協定驅動程式的實作最主要分成兩個部分，一是協定驅動程式的實作，二是串流介面驅動程式的實作。協定驅動程式得實作讓我們可以透過 NDIS 的介面和底層的網路介面裝置的迷你連接埠驅動程式溝通。而透過串流介面驅動程式的實作，則是為了讓在應用層的動態函式庫能夠存取 CLM 驅動程式。

### 6.5.1 協定驅動程式的實作

協定驅動程式的實作其實和中介層驅動程式十分類似，差別在因為是協定驅動程式，所要實作的函式只有協定驅動程式介面的函式。底下針對比較重要的協定驅動程式

介面函式做個別的說明

- 協定驅動程式介面

ProtocolBindAdapter

當每一個網路介面卡被迷你連接埠初始化後，NDIS 都會主動來呼叫這個函式，來要求協定驅動程式去跟底層的迷你連接埠驅動程式做連結。而在這個函式裡面，最重要的是就是去呼叫 NdisOpenAdapter 去連結底層的網路界面裝置。

ProtocolStatus

層的迷你連接埠驅動程式通常會維護一些網路介面卡的狀態，當狀態有所改變時，它會去呼叫 NdisMIndicateStatus 來告知上層的協定驅動程式。而協定驅動程式就必須實做本處理函式，來接受這些狀態的改變。

### 6.5.2 串流驅動程式的實作

關於串流驅動程式的實作，我在前面的章節已經介紹過。這邊比較不一樣的是，因為 CLM 驅動程式可能同時有多個應用程式來開啟和存取，所以必須支援多重存取的機制，所以在 CLM_Open 介面函式的地方，必須做額外的處理，我們必須在開啟本體儲存一些資訊，來紀錄開啟的應用程式相關的資訊和識別。

# 第七章 結論和未來工作

## 7.1 結論

在這篇論文當中，我們成功利用我們的中介軟體整合了多異質的網路介面。透過我們的整合，多網路介面的行動裝置可以在異質網路間漫遊並做無間縫的換手。首先我們成功移植了原本在 Window XP 平台上的 *RIOMIP* 軟體到 Windows CE 平台上；另外，我們針對 *RIOMIP* 在效能上的缺點進行架構上的重新設計和實作，用以獲得在手持平台上較佳的效能。

另一方面，我們將跨階層網路設計的概念導入到我們的中介軟體中，提供一套開放式的應用程式開發介面提供在應用程的行動管理程式所使用，用來增加在多異質網路介面下行動管理的效率，減少換手的延遲。同時，這套應用程式開放介面也可以提供未來在應用層開發其他具有移動性支援的應用軟體所使用。

## 7.2 未來工作

論文中所實作的中介軟體目前是在裝置 Windows CE 作業系統的手持裝置實驗平台（Intel DBPXA250）上執行。在未來，我們將移植我們的中介軟體到以 Pocket PC 或 Windows Mobile 為作業系統的市售手持裝置上。

另一方面，我們可以用我們中介軟體所提供的應用程開發介面，來開發在多異質網路介面下的移動管理軟體或其他具異動性的應用軟體。

# 參考文獻

[1] Milind M. Buddhikot, Girish Chandranmenon, et al., "Design and Implementation of a WLAN/CDMA2000 Interworking Architecture", IEEE Communications Magazine, Volume 41, Issue 11, pp. 90-100, Nov. 2003

[2] Carneiro, G.; Ruela, J.; Ricardo, et al., **"**Cross-layer Design in 4G Wireless Terminals**"**, IEEE Wireless Communications, Volume 11, Issue 2, pp. 7-13, Apr 2004

[3] C. Perkins, Ed., "IP Mobility Support for IPv4," IETF RFC 3344, August 2002.

[4] Douglas Boling, Programming Microsoft Windows Ce .Net, Third Edition, Microsoft Press, 2003

[5] M. Buddhikot *et al.*, "Integration of 802.11 and Third Generation Wireless Data Networks," IEEE INFOCOM 2003, Apr. 2003

[6] S. Shakkottai, T. S. Rappaport, and P. C. Karlsson, "Cross-Layer Design for Wireless Networks," IEEE Communications Magazine, vol. 41, no. 10, pp. 74-80, Oct. 2003.

[7] Floroiu, J.W., Ruppelt, R., Sisalem, D., Voglimacci, J., "Seamless handover in terrestrial radio access networks: a case study", IEEE Communications Magazine, Volume 41, Issue 11, pp. 110-116, Nov. 2003

[8] Ahmavaara, K., Haverinen, H., Pichna, R., "Interworking architecture between 3GPP and WLAN systems**",** IEEE Communications Magazine, Volume 41, Issue 11, pp. 74-81, Nov. 2003