國立交通大學

資訊科學與工程研究所

碩 士 論 文

支援具網路感知應用程式的中介軟體之設計與實作

Design and Implementation of a Middleware for Network-Aware Applications

研 究 生：許凱程

指導教授：曾建超　教授

中 華 民 國　九 十 五　年 六 月

支援具網路感知應用程式的中介軟體之設計與實作
Design and Implementation of a Middleware for Network-Aware Applications

研 究 生：許凱程　　Student：Kai-Cheng Hsu

指導教授：曾建超　　Advisor：Chien-Chao Tseng

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

June 2006

Hsinchu, Taiwan, Republic of China

中華民國九十五年六月

支援具網路感知應用程式的中介軟體之設計與實作

研究生：許凱程　　指導教授：曾建超

國立交通大學資訊學院資訊科學與工程研究所

摘　要

本論文設計和實作出一個軟體發展架構，讓程式開發人員在此架構上可以方便地開發具網路感知的應用程式。

近年由於各種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皆已具備存取網際網路的能力。不僅如此，在這些裝置同時配備多種無線網路接取介面也成為潮流，包括有線的區域網路、無線的區域網路、GPRS，甚至是 PHS 或是 3G。當使用者在不同的網路漫遊時，行動裝置可能切換至異質網路，對應用程式而言，由於網路狀態(頻寬、延遲、網際網路位址等)不再和以往一樣的固定了，所以應用程式需要根據網路狀態來調整自己的行為模式。

而現今要開發具網路感知的應用程式可能需要應用程式自己週期性的發出系統呼叫，以取得底層網路的資訊 (例如 IP 位址、路由表狀態或連線狀況)，可是網路狀態的改變不會如此頻繁，而且頻繁的發出系統呼叫可能會造成系統多餘的負荷。但是如果發出系統呼叫的週期太長則會讓應用程式無法及時反應網路的改變。

此外，在多網路模組的裝置上，需要一個行動管理員根據網路情況負責換手決策的動作，而行動管理員的程式撰寫者通常都需要去處理協定堆疊的狀態，以往處理網路的程式碼會根據不同的系統而使用不同的系統呼叫，可是這樣對於程式開發人員而言，直接使用系統呼叫是很容易出錯的，而且撰寫好程式也不易移植，有可能同樣的換手策略，在別的系統上就要重寫了。

為了解決以上的問題，本論文提出了中介軟體的解決方法，並且在 Linux 系統上實作。在此中介軟體中我們定義了取得協定堆疊資料和改變協定堆疊內部狀態的程式界面，也提供了事件通知機制，讓應用程式可以在網路狀態改變時直接被通知。以在 Linux 系統上操作協定堆疊而言，程式開發人員使用本中介軟體提供的程式界面所需撰寫的程式碼比直接採用系統呼叫來的簡潔。此外，我們只需將本論文的中介軟體移殖到別的系統，對於完全採用本中介軟體所提供界面發展的應用程式，毋需作任何修改即可移植到其它的作業系統上。而架構在本中介軟體之上的應用程式只需修改該應用程式額外使用系統呼叫的部分就可以跟著移植到別的系統。

Design and Implementation of a Middleware for Network-Aware Applications

Student：Kai-Cheng Hsu　　Advisor：Dr. Chien-Chao Tseng

Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

ABSTRACT

In this thesis we design and implement a software framework for programmers to develop network-aware applications on mobile devices with multiple interfaces. As wireless network and mobile technologies advance, nowadays most mobile devices on the market, such as notebooks, tablet PCs, PDAs, or smart phones, may be equipped with more than one network interfaces, such as wired LAN, wireless LAN, GPRS, PHS or 3G adaptors. Therefore, a mobile device may attach to different networks as it moves. As consequence, the applications running on the mobile device may encounter a network environment that varies more than ever, such as changes in bandwidth, delays, or even IP addresses. In order to tackle such network fluctuations, network-aware applications that can adapt themselves to the changes in network connectivity have now become a major research topic in recent years.

A network-aware application may need to issue system calls periodically to retrieve lower-layer network information, such as IP

addresses, Routing Table Entries and Link Connection Statuses. However network environment statuses may not change frequently and short intervals between system calls may result in wasting system resources for redundant information. On the other hand, with long intervals between system calls, applications can not react to network changes promptly.

Furthermore, a mobile device with multiple interfaces needs a mobility manager to monitor interface connectivity statuses and perform handover decision accordingly. However, in order to acquire network statuses and conduct a handover, a mobility manager needs to use system calls to communicate with underlying network protocol stacks. The use of system calls not only is error-prone but also makes applications not portable. Therefore we need to rewrite the mobility manager for a different system even if it uses the same handoff policy.

In this thesis we present the design and implementation of a software platform for the development of network-aware applications on Linux. The platform adopts a middleware approach to the above problems. The middleware provides application programming interfaces for the applications to interact with the underlying protocol stack and interfaces to acquire network statuses and manage the interfaces. Besides, the platform also provides an event notification mechanism for an application to register interested events of network environment changes and for the middleware to notify the application immediately when an event of interest occurs. Our implementation results show that the codes of protocol stack interaction and interface control is much more concise when using our middleware than using the system calls in Linux directly.

Furthermore, by porting our middleware on another system, we can port an application program on the system without any modification to the application program if the application program uses solely the interfaces provided by our middleware.

誌 謝

首先我要感謝我的指導教授—曾建超博士，讓我作了許多計畫的實作經驗，讓我有了這篇論文的研究方向以及提供我一個良好且自由的研究環境，同時要向我的論文口試委員：曹孝櫟 博士與王讚彬 博士及紀光輝 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。此外，還要感謝實驗室的同學、學長姐以及學弟妹在我碩士生涯中給我支持及鼓勵，謝謝你們。

同時也感謝我的家人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，他們在我最困難的時候給我的支持是我奮鬥下去的原動力。

目 錄

圖 目 錄

表 目 錄

第 一 章 緒論

1.1 研究動機

隨著網路科技的蓬勃發展，以無線的方式隨時隨地存取網際網路的資料，已是目前隨處可見的應用。現今可搭載多個網路模組的手持裝置也日漸普遍，如一般常見的雙模 (WLAN、GPRS) 手持裝置。因此目前的環境已經可以讓使用者在不同的地點使用不同的方式連上網路。

爲了讓使用者可以方便地在不同網路間作切換，我們需要一個行動管理員來做智慧型換手決定。在以往行動管理員的程式開發者需要根據不同的系統撰寫額外的程式碼來處理路由表的設定、協定堆疊的設定及硬體的控制等…。而且這對程式員而言都是不小的負擔。

實際上這些機制都是可以抽離出來的，我們只需要給開發人員一個簡單易用的程式設計界面。讓底層系統相關的部份由中介軟體處理掉，如此行動管理員的開發人員就可以需專注於換手策略 (Policy) 的設計而不用擔心系統相關的控制程式碼是如何運作和撰寫。而且若可以用一個統一的應用程式界面將系統底層部分隱藏起來，對於程式開發者而言就不用花額外的時間去處理各種系統的相關程式碼，可以專注在本身程式邏輯的開發。若程式開發者又對該系統不熟悉，則程式出錯的機率將會大大的提高，使用中介軟體開發應用程式可以降低程式處理系統部分出錯的機會 〔1〕。

現今使用者已經可以在同質或異質網路間換手，我們也可以說現今的使用者比已往更具移動性。也因此使用者所處的網路狀態和已往只是固定使用一個網路大不相同了。所以應用程式需要對網路的狀態有所感知，如此應用程式才可以根據不同的網路狀態來調整本身的行爲，讓使用者有較佳的使用經驗。

有的應用程式會需要知道使用者是否切換了網路。如 VoIP 的程式，如果該 VoIP 程式可以知道使用者目前切換到別的網路，該 VoIP 程式就可以重新和通話方建立連線。而使用此 VoIP 軟體使用者就不用特別處理換網路時的動作而且也不會感覺到使用上的不方便。

FTP 客戶端程式也是另一個例子，在沒有網路的時候，它可以先暫停下載，等到有網路可用時，它就可以繼續下載原本沒下載完的檔案。

爲了使應用程式可以擁有這些特性，網路狀況的事件通知機制是必要的。因此我們也會設計和實作此機制。

1.2 研究目標

本論文的研究目標是發展一個應用程式開發平台，可以讓程式設計者使用該平台來開發具網路感知的應用程式。藉著本論文所發展的平台，可以加速程式設計者對於行動通訊裝置的網路介面管理、漫遊與具網路感知的應用軟體的開發。

該平台會以中介軟體的形式來開發，藉著中介軟體的形式，期望該平台可以作到與作業系統的獨立性。

爲了到達目標，以下列出各個子目標

- 應用程式設計界面的定義
  - 具可擴充性
  - 簡單易用的界面
- 事件通知機制
  - 事件的定義
  - 跨層的事件通知
  - 處理事件的方式

1.3 章節簡介

本篇論文在稍後的章節編排與簡述如下：第一章描述本篇論文的研究動機，以及本篇論文希望達到的目標。第二章爲背景知識概述，包含中介軟體簡介、跨層信號簡介、具網路感知應用程式介紹、程式化換手策略、多網路整合及 Linux 下使用者空間和核心空間的訊息交換機制相關討論及研究。第三章則爲介紹與本論文相關的文獻，包括目前正在制定的兩個標準 IEEE 802.21 及 Toward Open and Unified Link-Layer API，和本實驗室之前對於在 Linux 事件通知機制的相關研究。第四章接著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。第五章便介紹我們實作本系統的各個細節，包含了軟硬體架構、各元件互動機制，及核心修改部份。第六章將介紹我們所開發出來的中介軟體成果及貢獻。其中包括如何開發具網路感知的應用程式的範例，我們會舉出一個簡單的換手策略程式和兩個使用者常用的應用程式 VoIP 及 FTP。最後，第七章對本論文做出總結，以及未來的研究方向。

第 二章 背景知識介紹

2.1 中介軟體簡介 (Middleware)

根據〔1〕的定義，中介軟體屬於軟體層，它是藉於作業系統上層、應用程式下層間的軟體。

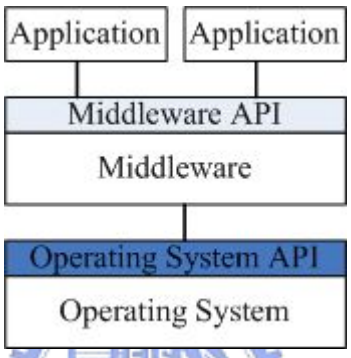


Figure 2-1 中介軟體概觀

對於程式設計者而言，中介軟體提供比作業系統更高階的系統函式和系統支援，因此使用中介軟體來開發程式有以下幾點好處：

- 所開發出來的程式更具移植性
- 讓程式設計員更有生產力
- 可以降低系統相關程式碼錯誤的機會

藉著中介軟體提供較高階的系統函式，中介軟體可以遮蔽許多系統上不同的異質性。

以下列舉出一些可以藉由中介軟體所遮蔽掉異質性的情況:

- 中介軟體可以遮蔽掉各種的網路存取技術
  - 如：數位電視的開發 (訊號可以透過各種不同的方式接收)

- 中介軟體可以遮蔽掉不同作業系統的異質性
  - 如：Java Virtual Machine
- 中介軟體可以遮蔽掉不同的程式語言
  - 如：Java RMI

本論文以中介軟體的方式來發展平台，其中一個目的就是爲了遮蔽掉不同作業系統的異質性。

程式開發人員可以使用以下各種不同的方法來使用中介軟體來開發軟體。

- 中介軟體提供函式庫和程式設計界面
  - 如：Linda、本論文提供的方法
- 從一開始就提供程式設計語言
  - 如：Java 和 Java Virtual Machine
- 提供介面定義語言 (Interface Definition Language) 用來映射到程式語言
  - 如: CORBA

接下來再來介紹各種不同的中介軟體類型：

- Distribute Tuples：(a, b, c, d, e)
  - Relational Database, SQL
  - Linda and Tuple spaces
- Remote Procedure Call (RPC)
  - 使得遠端的函式可以在本地端被查詢
- Message-Oriented Middleware (MOM)
  - 程式開發員藉著訊息和中介軟體溝通
  - 本論文所實作出的中介軟體屬於此類
- Distributed Object Middleware
  - 使得遠端的物件可以在本地端被查詢
  - CORBA

- DCOM/SOAP/.NET
- Java RMI

在〔2〕〔3〕〔4〕〔5〕〔6〕〔7〕都有提出以中介軟體來解決異質網路漫遊、換手和開發具網路感知應用程式的解決方案。

2.2 多個網路的整合

目前有愈來愈多的網路存取技術的出現，從早期的Ethernet有線網路到現今的802.11 無線區域網路，和個人行動數據服務的GPRS以及 3G、PHS等，及最近標準剛剛定好，產品正要開始上市的 802.16WiMAX無線網路。Table 2-1 顯示了目前常見的不同無線網路存取技術的標準、最大傳輸速率和使用頻帶。

Table 2-1 各種常見異質無線網路的相關資訊〔8〕

Network	Standard	Data rate	Frequency band
Cellular Networks	GSM data (2G) GPRS (2.5G) UMTS (3G)	9.6 kbps 14 – 128 kbps up to 2Mbps	900/1800/1900 MHz “ “
WLAN	IEEE 802.11b IEEE 802.11a IEEE 802.11g IEEE 802.11n	1, 2, 5.5, 11 Mbps 1 – 54 Mbps “ 100 – 540 Mbps	2.4 GHz 5 GHz 2.4 GHz 2.4, 5 GHz
Bluetooth	IEEE 802.15.1	721 kbps (BT 1.1) 2 – 20 Mbps (BT 2.0)	2.4 GHz
WiMAX	IEEE 802.16c	134 Mbps	10 – 66 GHz

現今無論是筆記型電腦，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搭配雙網甚至是多個網路模組的能力。但是目前的製造廠商都只是試圖將多種網路模組 (如 WLAN 和 GPRS) 加到裝置上，可是彼此的網路模組是互相獨立的，雖然隨著硬體環境的成熟，使用者可以順暢地在不同的網路下漫遊日趨可能。但目前缺乏網路模組自動交遞 (Handoff) 與管理機制，同時也未提供網路模組的互動機制，所以目前使用者還沒有辦法有效率地在不同網路間換手。因此藉著本論文所提出的開發平台可以幫助開發廠商快速地發展異質網路間的換手管理程式。

目前常見雙網手機就是無線區域網路與個人行動數據服務的整合架構〔9〕〔10〕，過去也有許多文獻在探討各種不同異質網路間的整合〔4〕〔5〕〔11〕〔12〕〔13〕〔14〕〔15〕。

Figure 2-2 顯示了一個搭載WLAN及GPRS網路模組的手機在異質網路間的漫遊。

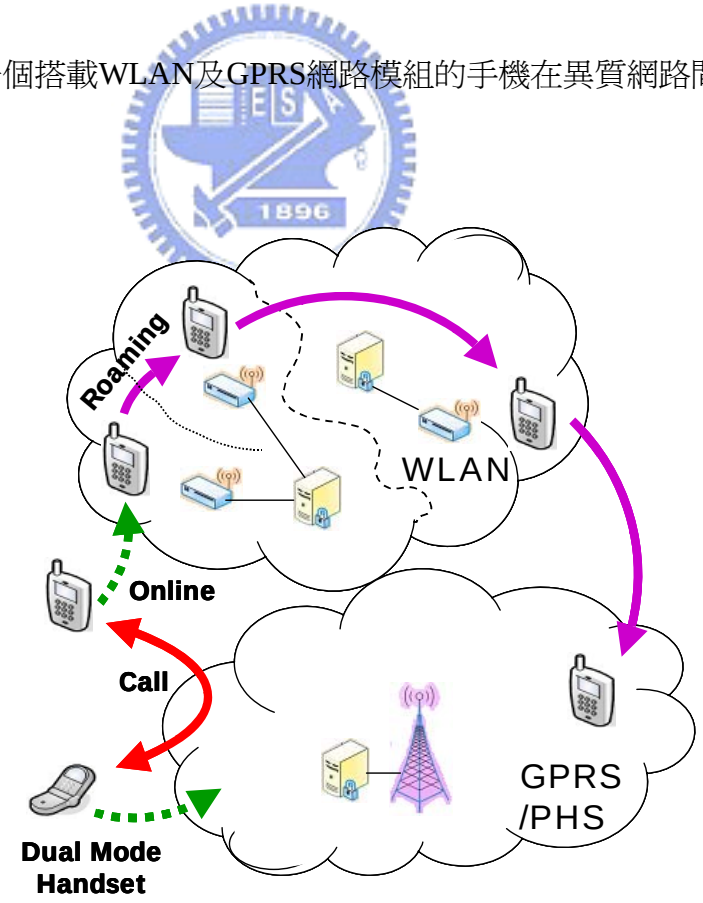


Figure 2-2 雙網手機漫遊示意圖

當使用者在不同的異質網路間換手時，會受到影響的就是各個網路應用程式，因爲在不同的網路間換手時，正在通訊的兩個應用程式間的會談 (Session) 會斷掉。所以在過去有各種方法被提出用來保持應用程式會談的持續性 (Session Continuity) 如SIP〔16〕及Mobile IP〔17〕。可是不論是 SIP 或著是 Mobile IP 都會遇到一個問題，就是負責掌控SIP或Mobile IP的程式如何得知使用者切換網路了呢？過去這一部份都是由使用者自行下達指令來告訴SIP作Re-Invite的動作或通知Mobile IP作Binding Update的動作，不過這顯然不符合使用者的方便性。因此本論文所提供事件通知機制可以幫助架構在SIP的程式得知使用者切換網路的事件後馬上作Re-Invite的動作，或是控制Mobile IP的Daemon得知使用者切換網路的事件後直接作Binding Update的動作，對於多個異質網路的整合有極大的幫助。

2.3 跨層設計簡介 (Cross Layer Design)

Figure 2-3 顯示出一個傳統的無線行動網路的架構。在此架構之下，標準的網路堆疊協定 (e.g. TCP/IP〔18〕) 會被實作在各個點上 (e.g. Base station、Access Point、Mobile Device) ，以此保證該系統可以在目前已發展成熟的網路上使用。

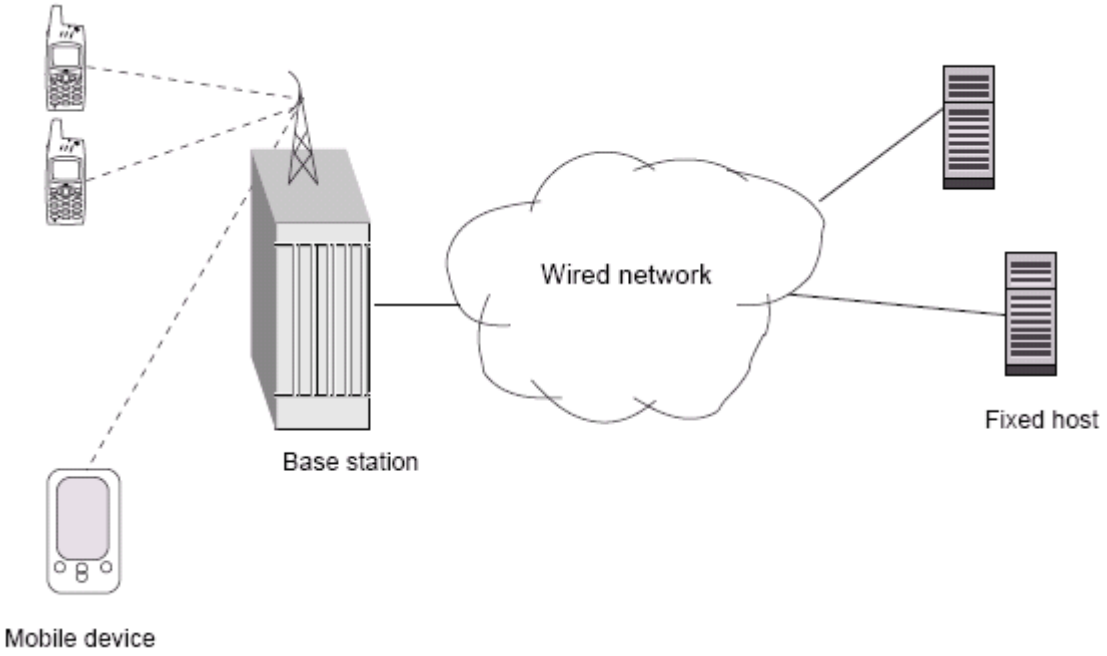


Figure 2-3 傳統的無線行動網路架構

參考 Figure 2-4 傳統的協定堆疊是以階層 (layer) 的方式設計和實作的，藉著分層可以清楚地將各層需要實作的功能抽象分離出來。因此各層間不需互相知道其它層的狀態，只要把自己該層的事情作好就行了。

Application	User Programs, interface, Higher layer protocols
Transport	Connection management, flow control, end-to-end layer
Network	Routing, Addressing
Data Link/Mac	Error free transmission, medium access
Phyical	Transmission of raw bits

Figure 2-4 TCP/IP protocol stack

但在〔19〕中有提到依據分層所設計的協定堆疊在無線行動網路的環境下運作起來是沒有效率的，這起因於無線網路連結的不穩定性及行動裝置天生就沒辦法搭載太多的資源。

在過去已經有許多想要藉由跨層回饋 (Cross Layer Feedback) 來增進目前協定堆疊效能的研究。如早期的研究在有線網路上〔20〕〔21〕顯示在不同的協定層中傳遞資訊對增進網路效能是十分有效。

跨層回饋的機制是指在協定堆疊中每個不同的協定層間可以互相的互動，以下舉出幾個例子。

- TCP 層可將封包遺失的資訊告知 Application 層，這樣 Application 就可以根據這些資訊來調整自己發送封包的速率。
- Link/MAC 層可以調整 Physical 控制的 power 大小來控制 bit-error rate。
- Link 層重傳封包的次數可以當作是目前 Channel 的狀況的測量，TCP 層可以藉著該資訊調整 retransmission timer。
- Application 層也可以藉著 Link 層得知目前 Link 的好壞來調整封包發送速度率。

在〔22〕篇文章中提到在目前現有在使用的協定堆疊以及網路架構都沒有納入多重介面的特性作爲考量，在該篇文章之中也提及，在進行垂直換手的時候(Vertical Handoff) ，由於不同網路接取技術的特性相差很多，因此會對系統效能造成很大的傷害。由於如此，該篇文章認爲應該存在著跨層 (Cross-Layer) 的通知機制，使得驅動程式的事件可以傳達至協定堆疊內的各層，如 Figure 2-5。Figure 2-5 顯示了許多資訊可以在協定堆疊內的不同層中傳遞，如Security、QoS、Mobility及Wireless link adaptation。

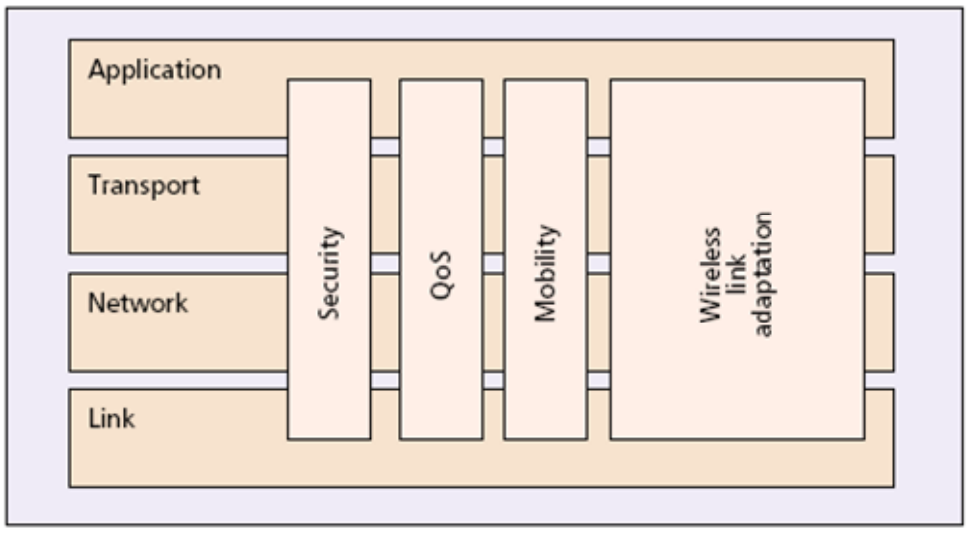


Figure 2-5 跨層設計概念示意圖〔22〕

行動漫遊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。而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。然而這類行動管理程式通常是使用者空間的應用程式，傳統上，介面管理程式會週期性的發出系統呼叫，以取得底層介面的資訊 (例如網路介面的種類、頻寬或連線狀況)。但是，由於網路介面的狀態改變可能不會如此的頻繁，因此頻繁的發出系統呼叫可能會造成系統多餘的負荷。此外，在需要快速換手的情況下，行動管理程式必須對網路介面的狀態改變做出迅速的反應，而在傳統的狀況下，網路介面狀態更新的頻率取決於程式發出系統呼叫的頻率，發出系統呼叫的頻率愈高，就愈容易得到即時的資訊，當然對系統的負荷也就愈大。

因此本論文導入了跨層網路設計的概念，能提供底層網路資訊給使用者空間的行動管理程式或其它應用程式，並透過我們設計的事件觸發機制，讓相關程式能夠快速地被告知底層網路的改變，並做出適當的反應，進而提供良好的服務品質。

2.4 具網路感知應用程式簡介

可以根據目前所處網路狀態來調整本身行為的應用程式稱為具網路感知的應用程式。

具網路感知的應用程式也算是一種跨層設計的應用，只是把協定堆疊各層的資訊都傳達給應用層。而用來處理異質網路間換手的行動管理員程式也是屬於具網路感知的應用程式，因為行動管理員程式若是無法得知網路情況，則行動管理員程式如何作有效的換手？最低限度也必需要知道每張網路界面卡的連線狀態、連線品質、訊號強度等資訊，這樣行動管理員才可用取得的資訊來做決定要使用哪個網路。

傳統的網路程式設計是不知道底層的網路狀態的，所以我們會常常看到 FTP 或 Email 的程式，在連線失敗後會設一個計時器，等到時間倒數完了再試一次，是嘗試及錯誤 (try & error) 的作法。但如今在多模行動裝置下，使用者也會比以往更具有移動性，因此再用過去的方式來設計網路應用程式，會顯得沒有效率。如果在一開始設計網路應用程式的時候就納入網路底層狀態的考量，相信所開發出來的程式會更適合以後擁有高度移動性的使用者。如影像串流客戶端 (Video Streaming Client) 的應用程式如果有辦法得知目前連線網路的頻寬狀態，該客戶端應用程式就可以通知影像串流伺服器 (Video Streaming Server) 根據客戶端目前的頻寬來調整撥放的品質。

現今也有許多關於具網路感知應用程式方面的研究文獻，可參考〔3〕〔23〕〔24〕〔25〕〔26〕〔27〕〔28〕。

2.5 Linux 下使用者空間和核心空間的訊息交換機制

核心可以通過各種不同的方式將核心內部的訊息提供給使用者程式。本節將介紹 Linux 系統下使用者程式與核心溝通的機制。

2.5.1 procfs (/proc filesystem)

這是一個虛擬檔案系統，通常掛載在 /proc 目錄下。核心透過檔案的方式將核心內部訊息傳遞給使用者空間的程式。因爲是以檔案的形式存在，因此我們可以用 cat 或著是 more 指令來讀取核心訊息。甚至可以用導向符號 “>”將訊息傳遞給核心。

2.5.2 ioctl 系統呼叫

iotcl (輸入/輸出控制)系統呼叫可以用來操作一個檔案描述子。ioctl 通常用來作一些設備的特殊操作。ioctl 也可以用來操作 socket 描述子來控制網路，在一些網路管理程式上就是使用 ioctl 來操作 socket 描述子來控制網路，如 ifconfig 和 route 等。

讓我們以 ifconfig 爲例子。ifconfig 就是使用 ioctl 與核心溝通的。如果系統管理員輸入命令 “ifconfig eth0 mtu 1250” 來修改 eth0 設備的 mtu。ifconfig 首先打開一個 socket，並根據系統管理員的輸入初始化一個數據結構，然後把這數據結構傳給 ioctl。修改 mtu 的命令爲 SIOCSIFMTU。

Table 2-2 使用 ioctl 操作網路設備的 MTU

```
struct ifreq data;
fd = socket(PF_INET, SOCK_DGRAM, 0);
/* 初始化 “data” 資料結構，用於改變 MTU */
err = ioctl(fd, SIOCSIFMTU, &data);
```

接著介紹 ioctl 的命令規則，如在路由表新增路由的命令，SIOCADDRT，可以被分解爲 SIOC ADD RT。它強調兩件事情：ADD 表示要添加一些東西，而 RT 表是添加的爲路由表內容。大多數的命令都遵循這個規則。有時，如果一個對象既可以讀，又可以寫，相對應的命令名字就會有多個選項：G 表示 GET (讀取)、S 表示 SET (設置) 。如在網路卡上取得和新增 IP 位址的命令就分別是 SIOCGIFADDR 和 SIOCSIFADDR。

與網路相關的 ioctl 命令的定義在 include/linux/sockios.h 中。設備驅動程式也可以自己定義新的私有命令，命令編號的範圍從 SIOCDEVPRIVATE 到 SIOCDEVPRIVATE+15。

2.5.3 Netlink socket

RFC3549〔29〕所描述的Netlink socket是使用者空間的程式和核心交換訊息也是最常用的方式。Netlink socket 可以使用標準的 socket API開啓、關閉，接收和發送訊息。

讓我們來看一下 socket 系統函式的原型：

int socket(int domain, int type, int protocol)

domain 參數指定了 socket 是在哪個 domain 下進行通訊。

type 參數通常爲則指定了該 socket 資料傳輸的類型，如 SOCK_STREAM 提供了有順序的、可靠性、有雙方連線基礎的資料傳輸類型，而 SOCK_DGRAM 則提供無順序、不可靠的資料傳輸類型。

protocol 參數指定了在該 domain 下的 socket 是要採用何種協定。

Netlink 使用PF_NETLINK domain，type 爲 SOCK_DGRAM，並且定義了幾種不同的 protocol，每一種 protocol都會對應到不同的核心組件。如 NETLINK_FIREWALL被防火牆所使用。Netlink 的 protocol列表在 include/linux/netlink.h文件中定義，名稱爲 NETLINK_XXX。Table 2-3 顯示了一部分的定義。而之後會介紹的rtnetlink則爲 NETLINK_ROUTE的protocol。

Table 2-3 include/linux/netlink.h 中關於 protocol 的定義部份

#define NETLINK_ROUTE	**0**	/* Routing/device hook */
…		
#define NETLINK_FIREWALL	3	/* Firewalling hook */

如同一般的 IP 訊息有 IP 表頭一樣，Netlink 的訊息也有 Netlink 的表頭。

Figure 2-6 Netlink Message Format

Table 2-4 顯示了 netlink 的表頭的定義，這邊比較重要欄位的是 nlmsg_type，這個 2 bytes 的欄位定義了接在netlink表頭後面資料要如何解釋。

Table 2-4 netlink 表頭的定義

```
struct nlmsghdr {
        u32 nlmsg_len;      /* Length of message including header */
        u16 nlmsg_type;     /* Message content */
        u16 nlmsg_flags;    /* Additional flags */
        u32 nlmsg_seq;      /* Sequence number */
        u32 nlmsg_pid;      /* PID of the process that opened the socket */
};
```

在 netlink socket 中，所要傳送訊息都需要指定端點的 id，而端點 id 通常是用打開 socket 的使用者程式的 PID (process identification) 來表示，而 0 通常就代表核心。

netlink 的一個特性就是可以發送單點和廣播的訊息：即目標端點可以是使用者空間的程式的 PID 或著是廣播的 id。Linux 核心中已經定義了幾個廣播的 id，用於發送特定事件的通知，如果使用者空間的程式對某類事件感興趣，可以把 PID 註冊到相對應的廣播 ID 中。

廣播ID的定義在 include/linux/rtnetlink.h中，名字類似 RTMGRP_XXX，可參考 Table 2-5。其中RTMGRP_IPV4_ROUTE和RTMGRP_NEIGH兩組廣播ID，分別用於通知路由表和L3 及L2 的更改資訊。

Table 2-5 include/linux/rtnetlink.h 關於廣播事件的定義部份

```
#define RTMGRP_LINK                1   /* Link 相關事件 */
#define RTMGRP_NOTIFY              2   /* notify chain 相關事件 */
#define RTMGRP_NEIGH               4   /* 附近鄰居相關事件 */
…
#define RTMGRP_IPV4_IFADDR         0x10   /* IPv4 Address 相關事件 */
#define RTMGRP_IPV4_ROUTE          0x40   /* 路由表相關事件 */
…
```

rtnetlink 是架構在 netlink 上的訊息，要使用 rtnetlink 的訊息就是在開 netlinksocket 時以 NETLINK_ROUTE 作爲 protocol 的參數。

rtnetlink 也有自己的訊息分類，每個分類都有自己的屬性 (Attribute) 資料，不同的屬性資料需要根據目前所收到的 rtnetlink 訊息爲哪一類來解釋。每個屬性資料前面也會有個屬性表頭，用於告知後面接的屬性資料的解釋。

Table 2-6 顯示了屬性表頭的定義，rta_len表示後面接的資料長度，rta_type表示要如何解釋後面接的屬性資料。

Table 2-6 屬性表頭 ratttr 的定義

```
struct rtattr {
        unsigned short rta_len;
        unsigned short rta_type;
}
```

而目前收到的訊息類型就定義在 netlink 表頭的 nlmsg_type 欄位中。以下列出目前 rtnetlink 所擁有的部分訊息分類：

- LINKS
  - 關於網路界面卡的訊息
- ADDRESSES

  關於網路界面卡上 IP 位址的訊息
- ROUTES
  - 關於網路路由表的訊息

- NEIGHBORS
  - 關於 ARP Table 的訊息
- RULES
  - 關於路由規則的訊息
- DISCIPLINES
  - 網路佇列相關訊息
- CLASSES
  - 網路傳輸分類相關訊息
- FILTERS
  - 網路傳輸過濾器相關訊息

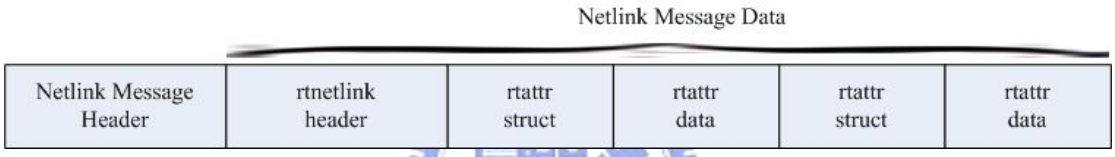


Figure 2-7 rtnetlink 訊息格式

Figure 2-7 顯示了 rtnetlink架構在netlink上的訊息格式，我們可以看到它就只是一連串帶著屬性的資料而已。而前面有一個 rtnetlink 的表頭。rtnetlink 的表頭也是會根據不同的rtnetlink訊息類型而有不同的解釋方式。

本論文的事件通知機制和系統有關的部份就是架構在 rtnetlink 下，我們去擴充 Linux 核心下的 rtnetlink 功能來達到我們所定義的各種事件通知。

與 Linux 其它使用者空間與核心空間的訊息交換機制相比，比如和 ioctl 相比，netlink 的一個優勢就是可以初始化一個傳輸過程而不僅僅只是對使用者空間程式的請求返回應答訊息。在 netlink 下，使用者空間應用程式和核心層間是有連線狀態存在的，所以核心空間有訊息產生時可以直接傳送給使用者空間，而 ioctl 的呼叫返回模式是沒辦法作到由核心主動傳送訊息給使用者空間程式的。

第 三章 相關研究

3.1 IEEE 802.21 Media Independent Handover Services

關於本節更詳細的內容可以參考〔30〕。

在 2004 年 3 月 IEEE 802.21 工作小組正式成立，802.21 是由 Intel 所主導之規格，其目的爲使行動裝置在不同網路之間漫遊時，提供相關異質網路技術界接，且能自動選擇最好用的網路連接類型，並無縫隙地切換通訊路徑。

802.21 之核心在於 Media Independent Handover Function （MIHF），其主要功能在將資料連接層中之 802.11，802.16 或 Cellular 等不同無線技術之 MAC、PHY，用一共同介面將資料成功切換至網路架構中 Layer 3 以上之層級。

Figure 3-1 顯示整個 802.21 整體架構，MIHF共定義了三個不同的服務：Media Independent Events Service (MIES)、Media Independent Command Service (MICS) 與Media Independent Information Service (MIIS)

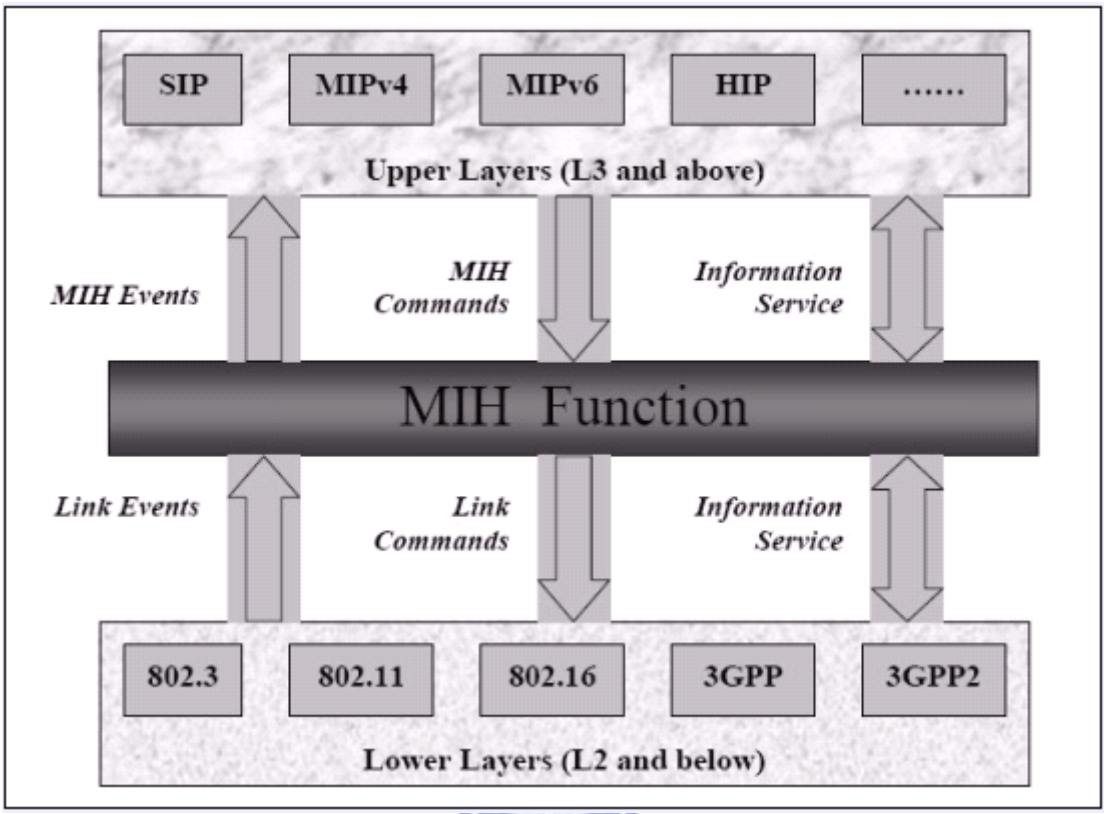


Figure 3-1 802.21 Overview〔30〕

整個 MIH Function 運作，首先在使 Layer2 之不同網路技術透過 MIES 向上傳遞，而上層如 SIP、IPv4、及 IPv6 等也可透過 MICS 往下層傳達命令，最後透過 MIIS 提供 MIFH 實體層能夠發現在搜尋範圍內之訊息。

透過 802.21 能讓聲音 (Voice) 及影像 (Video) 等講求及時 (Real-time) 之應用可依環境所需速度適時切換至所需網路，而不產生訊號延遲現象。

目前在 IEEE 802 之各種傳輸技術中，並沒有無縫隙界接之機制，因此在 802.11 技術已成熟及 802.16 技術也已問世，802.21 所扮演的角色將趨於重要。將 Wi-Fi、WiMAX 及 WWAN 之間透過其 MIH、Information Service 及 Smart Trigger 等三大 802.21 重要功能，順利將異質網路間訊號順利串接，也將左右整合三大網絡之終端產品發展。

3.2 Gollum (Toward Open and Unified Link-Layer API)

關於本節更詳細的內容可以參考〔31〕及 http://www.ist-gollum.org/。

GOLLUM (Generic Open Link-Layer API for Unified Media Access) 是從 2004 年 9 月開始，目前仍在進行中的研究計畫。這個計畫的目標在於發展創新的與網路程式相關的中介軟體、及程式介面，用來解決應用程式設計在異質無線行動網路下所衍生的問題。

GOLLUM 計劃研究和設計一套公開的、獨立於作業系統的 Link-Layer API (ULLA; Universal Link Layer API)，用來統一應用程式及作業系統存取和設定不同有線及無線網路的介面。有了統一的 link-layer 存取介面，應用程式設計將不再需要熟悉大量現存不同無線攫取技術的 API，對於行動網路應用程式的開發將會是一大福因。統一的應用程式設計介面，也會增進異質無線網路使用之間的相容性和也會使應用程式在不同無線攫取技術下移植更容易進行。GOLLUM 計劃的目標便是設計一套具有彈性和新型提供行動相關資訊和事件功能的應用程式介面，用來簡化無線網路應用程式的開發，提供未來更具智慧和創意的無線應用。

由本節和上一節的介紹，我們可以知道在未來行動裝置可以搭載各種不同的網路模組的趨勢下，各大廠商都積極發展獨立於不同網路模組的應用程式界面，用來將網路界面卡的異質性隱藏起來，並且也提供事件通知機制，讓上層的協定可以快速地知道底層網路界面的狀態。

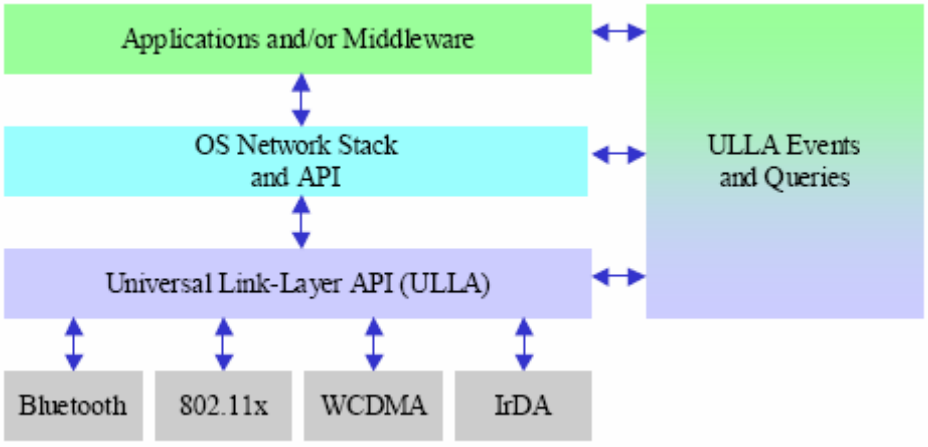


Figure 3-2 ULLA API 和相關事件查詢、中介軟體關係圖示

Figure 3-2 顯示了各種不同的無線網路存取技術可以經由 ULLA 整合起來。

Gollum 和 802.11 一樣也都處於發展階段，目前只有提出一個整體的大架構，對於程式界面或著 Link Layer 的事件都還未有詳細定義的文件。

3.3 Linux 平台上驅動程式層網路事件通知機制

〔32〕是本實驗室之前所發展的系統。該系統將網路介面相關的狀態改變利用類似傳統UNIX作業系統上的信號 (signal) 機制通知使用者空間的程式，並且修改了排程的演算法，使得使用者空間程式不必一再地發出系統呼叫，加快對於網路介面狀態改變的處理，以增進系統效能。

該系統之實作可以分爲三個主要部分：

- 事件通知部分 (Event notification)
- 行程管理 (process management)
- 排程器 (scheduler)

當網路驅動程式相關訊息發生以後，該系統的目標是由核心空間直接切換至使用者空間來執行。也就是說，在行程從核心空間回到使用者空間之前，該系統會去檢查是否有該行程所註冊的事件發生，當有註冊的事件發生時，系統會先執行該行程所註冊的對應處理函式，接下來，利用系統呼叫返回核心空間，並且在該系統呼叫內回覆原來所應該執行行程的硬體環境 (hardware context) 。

關於使用者空間和核心空間的切換，該系統利用了 Linux 下的兩種堆疊 – 核心模式堆疊 (kernel mode stack) 和使用者模式堆疊 (user mode stack) ；核心模式堆疊主要作用有兩點，記錄使用者硬體環境 (hardware context) 以及存放程序敘述子。

當要跳至使用者空間執行行程註冊的回叫 (Callback) 函數之前，因爲在 Linux 中每次由核心空間返回使用者空間都會將核心模式堆疊清空，如此一來，當使用者由回叫函數返回核心空間時，原本用以執行一般程式碼的硬體環境會消失；因此該系統將核心模式堆疊中的硬體環境存至別的地方，以免原本的硬體環境消失，因此，該系統將原本的硬體環境儲存在使用者模式的堆疊中，等到由回叫函數返回後，再將其復原。

系統核心內的關係如 Figure 3-3 所示。當網路事件發生時，通用驅動程式 (Generic Driver Layer)會去呼叫 Event Handler，Event Handler 就會去更改核心用來處理程序的資料結構，通知有事件發生，當排程器排到該程序時會先檢查是否有事件發生，有的話，就會去執行程序所註冊的回叫函式。

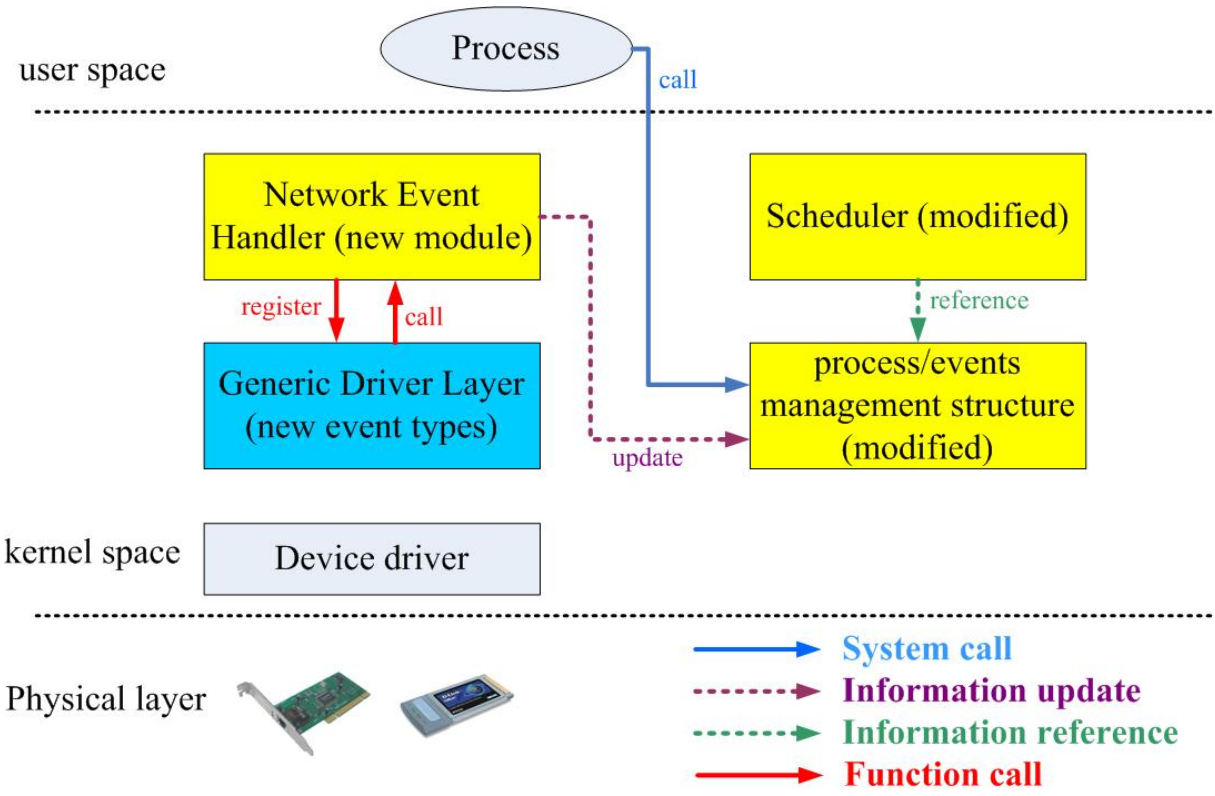


Figure 3-3 Linux平台上驅動程式層網路事件通知機制的系統元件架構和關係〔32〕

第 四 章 支援網路感知之中介軟體設計與架構

本章將介紹本論文所提的支援具網路感知應用程式的中介軟體設計架構。我們將此中介軟體命名爲 WinME。

4.1 目標與問題定義

本論文的主要目標是發展一個應用程式平台，讓程式設計者可以用來開發具網路感知的應用程式，因此我們有以下的子目標：

- 應用程式可以取得網路資訊
  - 應用程式可以主動要求網路相關資訊。
- 當網路狀態改變時，應用程式可以被通知
  - 應用程式可以被動的被通知事件的產生，不需要去使用系統呼叫輪詢網路狀態是否改變。
- 程式設計者可以只註冊自己感興趣的事件
- 程式設計者開發網路程式時不需擔心網路相關的系統程式碼
- 行動管理員程式可以控制協定堆疊各層的資訊 (e.g. 設定 IP 位址等…)

爲達以上的目標，我們會有以下的問題：

- 應用程式如何取得網路資訊？
- 如何提供事件驅動機制？
- 程式界面的定義。
- 如何去控制協定堆疊和網路界面卡？

接下來的章節將會介紹我們的系統如何完成以上的目標及解決上面所提及的問題。

4.2 系統概觀

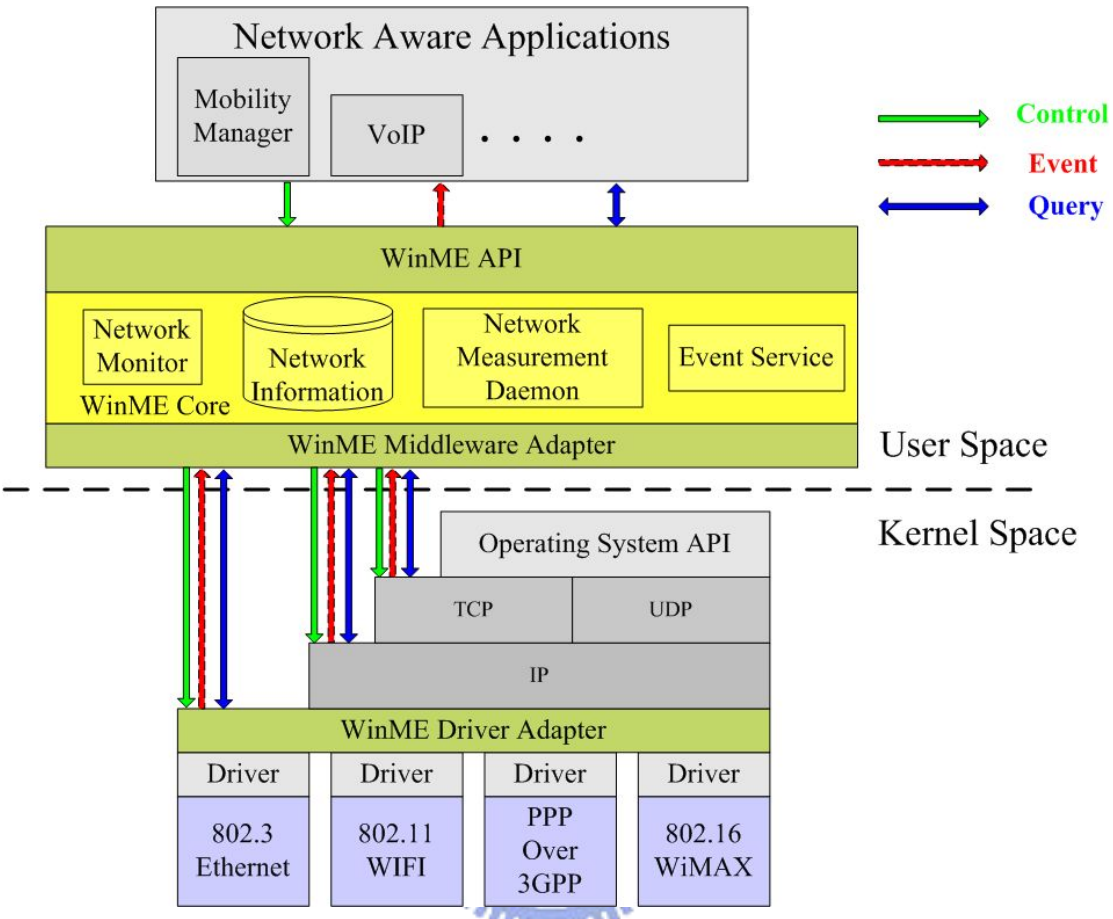


Figure 4-1 中介軟體概觀

Figure 4-1 顯示了整個WinME的架構，我們可以很清楚看到應用程式透過WinMe所提供的程式界面去對系統底層的網路作存取的動作。這邊也有跨層設計的概念，我們將位於應用層下層協定堆疊的資訊直接傳達給使用者空間的應用程式，使用者空間的應用程式也可以直接將資訊傳遞給協定堆疊的底層。

我們將這些資訊都拉上使用者空間的應用程式的好處是我們在控制底層網路時可以有較大的彈性，完全取決於程式設計員的設計方式。以行動管理員來說，行動管理員的決策很可以很複雜的，也可能會常常更動，這時候要求效率而把換手決策作進核心內就顯得不適當。

當一個 802.11 的網路界面卡和無線接取點連上線後，會產生一個 Link UP 的事件，並且會帶上 ESSID 等資訊，我們知道當連結上之後需要給網路界面卡設定 IP 位址。

讓我們來考慮兩種情況：

一種情況是 802.21〔30〕的MIH Function，在這個情形之下Link UP的事件會傳達給網路層(Network Layer)，此時網路層也許就直接把IP位址設定到網路界面卡上。

另一種情況就是使用 WinME，在這個情形之下，Link UP 的事件會傳達給應用程式，應用程式經過決策後，可能決定使用 DHCP 取得 IP 位址，或經由使用者設好的 Profile 決定 IP 位址，然後再把 IP 位址設定到網路界面卡上。

很明顯地，第一種情況的效能會比較好，可是沒有彈性，而今 802.21 的也尚未成爲標準，而且要達到第一種情形，我們還必須更改核心中關於協定堆疊的程式碼，使其可以支援 MIH Function。我們認爲關於換手的策略應該在使用者空間的應用程式作決定，本論文並不關心行動管理員程式是如何作換手決策的，本論文只是提供一個程式開發平台，讓行動管理員程式的撰寫者可以方便地開發和處理網路間的換手策略。本論文提供的只是一個機制 (Mechanism)，而程式的策略 (Policy) 就交給應用程式的開發者。

接下來的小節中，我們會介紹 WinME 中的各個元件。

4.2.1 控制 (Control) 、查詢 (Query) 及事件 (Event) 介面

應用程式透過 WinME 所提供的應用程式介面來使用控制、查詢和事件介面。

控制界面是用來控制協定堆疊或網路界面的狀態，如設定網路位址、增加路由表欄位、更改預設閘道或叫無線網路卡去掃描鄰近的接取點 (Point Of Attachment) 等…。

查詢界面則提供應用程式用於查詢目前的網路資訊，如連線訊號強度、目前網路頻寬、網路卡的狀態等。像 VoIP 的程式在一開始啓動時可能會需要知道目前連線網路卡的 IP 位址，然後再用此 IP 位址去 SIP Server 註冊。

事件介面將網路事件提供給應用程式，如 Link UP、Link Down、IP Change 等事件。這對網路換手的效能很重要，有了這個介面之後應用程式就不需要去輪詢底層的網路狀態來判斷事件是否發生。

4.2.2 行動管理員 (Mobility Manager)

使用 WinME 所提供的程式界面來管理底層的硬體或網路狀態。也因爲如此行動管理員的設計者可以只需要專注於換手策略 (Policy) 的設計，而無需去關心底層是如何的運作。

行動管理員也可能包含使用者圖形界面 (GUI) 讓使用者來設定換手策略相關的資料，或顯示出目前網路狀態。

最簡易的雙網 (802.11 WLAN and GPRS) 行動管理員，它可能是以無線區域網路作爲連線優先，當有無線區域網路時，就將預設閘道設成無線區域網路下的預設閘道，在沒有無線區域網路訊號的範圍下，就將預設閘道設成個人行動通訊 GPRS 下的預設閘道。所以我們是藉著更改預設閘道來達到選擇網路界面卡的效果。

4.2.3 WinME 的應用程式界面 (Middleware API)

讓程式開發者使用這些界面來與 WinME 溝通，協助開發應用程式。

4.2.4 WinME 的核心元件

本節會介紹 WinME 核心中的各個元件說明。

4.2.4.1 網路資訊儲存體 (Network Information Storage)

用來存放所量測的網路資訊，如網路頻寬、訊號強度、網路界面卡的 IP 位址等。雖然資訊有可能會過時，不過在當有很多不同的應用程式來查詢同一份資料時，WinME 就可以快速地回應應用程式。有的資訊可能會常常變動的 (如訊號強度)，我們就會設一個新鮮時間 (Fresh Time)，如果應用程式下達取得訊號強度的資訊，而這個資訊又過期的話，WinME 就會去跟底層取得資訊。至於不常變動的資訊 (e.g. IP 位址)，則 WinME 會在收到底層 IP 位址改變的事件時，就會更改相對應的資訊。

此元件存在的目的就是爲了快速地回應使用者程式的查詢要求，使得 WinME 不需要每當使用者程式要求一個網路資訊時，都要去跟系統底層要求一遍。

4.2.4.2 網路觀察者 (Network Monitor)

週期性的去觀察網路狀態，如訊號強度、網路卡資訊等資料….。觀察到的結果會存入網路資訊儲存體中。

4.2.4.3 網路測量程式 (Network Measurement Daemon)

可產生封包來測量目前的網路頻寬。測量的結果會存入網路資訊儲存體中。此元件在雖然在本論文中有納入WinME的設計，可是在實作上本論文並沒有實作該元件，因爲關於頻寬的測量本身就是一個值得討論的研究議題〔33〕〔34〕〔35〕〔36〕。或許可以在未來實作出這一個元件。

4.2.4.4 事件服務元件 (Event Service)

此元件提供應用程式事件處理機制，和處理底層的網路事件。應用程式可以透過 WinME 的程式界面來跟此元件註冊感興趣的事件，而網路底層的事件會先通知事件服務元件後，然後再由該元件再將訊號分送給感興趣的應用程式。

4.2.4.5 中介軟體轉接器 (Middleware Adapter)

這部份實作了所有和作業系統有關的程式碼 (如系統呼叫)。可用於幫助整個 WinME 的移植(porting)。

4.2.4.6 驅動程式轉接器 (Driver Adapter)

定義好驅動程式要提供的界面，讓 WinME 可以直接使用這個界面來對網路硬體作溝通。這有點像是 802.21 的 MIH 提供的部分機制，不過在 802.21 尚未成爲標準前，我們只能自己先這樣作，如果未來 802.21 成爲標準，而且核心內也實作的話，驅動程式轉接器有機會用 802.21 取代掉。

4.3 網路事件通知機制

本論文所提出的系統中，事件通知機制可以說是十分重要的部份。有了這個機制，應用程式就可以快速地反應網路狀態的變化，也不用像過去傳統的設計一樣，週期的使用系統呼叫來輪詢網路狀態，造成系統不必要的負擔。

由於應用程式並不會對每個網路事件都感興趣，所以 WinME 提供讓應用程式註冊事件的功能。有了註冊事件的功能之後，接下來是當事件發生時應用程式如何得知？WinME 提供兩種得知事件的方法，我們把他們分別稱爲同步處理和不同步處理。

在同步處理事件的架構下，網路事件發生時，會被送到應用程式的事件佇列中。而程式在執行的過程需要自己去檢查是否有網路事件的發生，有的話再進行事件的處理。雖然這也算是一種輪詢機制，不過這是只有在使用者空間的應用程式層去輪詢是否有事件發生，和以往下達系統呼叫輪詢網路界面的資訊在意義上是完全不同的。

在非同步的處理事件架構下，應用程式在和WinME註冊事件時需要提供事件處理者 (Event Handler)，此時WinME會幫該應用程式開一個執行緒並且用應用程式所提供的事件處理者來處理事件。所以原本的應用程式可以專注於本身的工作，當有事件產生時，對應的執行緒就就被喚醒起來處理該事件。這種處理事件的方式類似之前本實驗室所提的以信號方式處理事件的系統〔32〕。

接下來介紹 WinME 對事件通知的設計概念。

當程式設計人員想要使用 WinME 的事件通知機制時，需先作初始化的動作，在作完初始化的動作後，WinME 會配置一個事件佇列 (Event Queue) 給應用程式，並且返回一個事件描述子 (Event Descriptor) ，所以每個事件描述子都對應著一個事件佇列，應用程式藉著事件描述子來存取事件佇列，像是註冊感興趣的事件或檢查是否有事件發生。由以上的描述可以知道應用程式可以不止有一個事件佇列，應用程式可以有多個事件佇列，可能有的想要用非同步處理，有的想要同步處理，或著應用程式對於不同的事件要在不同的檢查（如在程式中某個點檢查 LinkDown 的事件，而再程式中另一個點檢查 IP 位址改變的事件）。

當應用程式作完初始化的動作得到事件描述子後，再來就是要註冊該事件佇列感興趣的事件，當網路事件發生時，也只有註冊過的事件才會被放到事件佇列中。

Figure 4-2 顯示了該應用程式有兩個事件描述子，事件描述子一註冊了 Link Up、Link Down、NETDEV_UP的事件，並且對應的事件佇列中有一個Link UP的事件未處理。而事件描述子二註冊了 IP Change及Default Gateway Change的事件，而該對應的事件佇列則有 IP Change事件未處理。

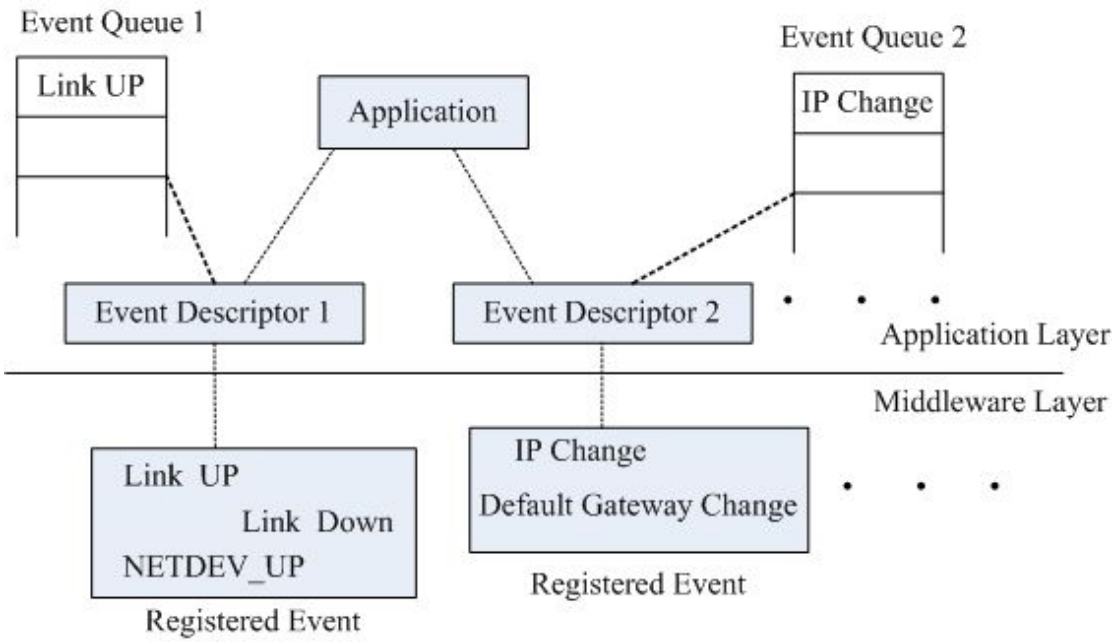


Figure 4-2 事件描述子和事件佇列等關系

接著介紹本論文WinME的設計如何作到之前所提及的同步和非同步的處理事件方法。應用程式決定要使用同步 (自行檢查事件的產生) 和非同步 (交由執行緒來處理事件) 在事件一開始初始化就決定好了。如果在事件一開始初始化的時候有提供事件處理者，就表示該事件描述子所註冊的事件都是採用不同步的方式作處理。反過來說，如果一開始初始化事件的時候，沒提供事件處理者的話，就表示該事件描述子所註冊的事件都是以同步的方式作處理。Table 4-1 和 Table 4-2 分別顯示了兩種處理事件方式的程式碼。

Table 4-1 同步事件處理的程式架構

```
/* event initialization, get event_descriptor */
/* win_event_init ‘s param is NULL, means synchronous mode */
event_descriptor =    win_event_init(NULL);
/* register interesting events */
win_event_register(event_descriptor, NETDEV_UNREGISTER);
win_event_register(event_descriptor, IP_CHANGE);
…
/* check if events happen */
if(win_check_event(event_descriptor, wevent) == R_HAVE_EVENT)
..
```

Table 4-2 非同步事件處理的程式架構

```
/* Event Handler */
/* When event happen, thread will call this procedure */
void my_event_handler(struct win_event* wevent) {
        printf("Event Happen!!\n");
}
int main() {
     …
     /* Programmers don't need to check if event happen*/
     event_descriptor =    win_event_init(my_event_handler);
     win_event_register(event_descriptor, NETDEV_UNREGISTER);
     win_event_register(event_descriptor, IP_CHANGE);
     ….
}
```

本節的最後將介紹事件的派發過程。當系統底層網路事件產生時，全部都會送到WinME的事件服務元件 (Event Service)，然後事件服務元件再根據應用程式所註冊的事件，把事件傳送給各個應用程式。Figure 4-3 顯示了事件傳送的流程。

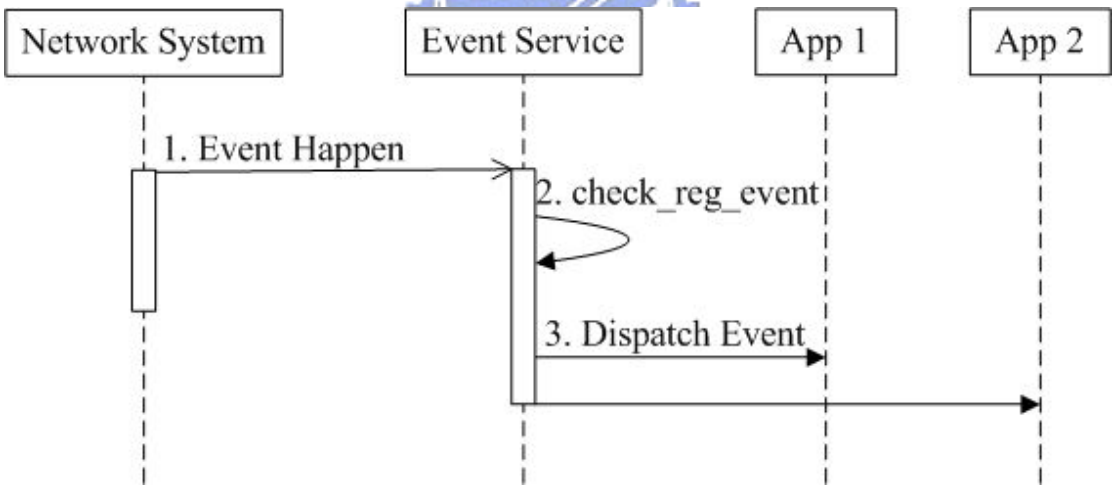


Figure 4-3 事件傳送順序圖

Figure 4-3 的各個步驟詳述如下：

Table 4-3 事件傳送順序圖的各步驟描述

步驟	描述
1. Event Happen	系統網路底層的有事件產生，將事件傳送給中介軟體內的事件服務元件。
2. check_reg_event	事件服務元件檢查哪些應用程式有註冊過該事件。
3. Dispatch Event	把事件傳達給有註冊的應用程式。

4.4 網路事件及命令定義

本節將簡單介紹本論文所提 WinME 所定義的事件和控制查詢命令。我們將事件和命令分成以下四大類：

- 系統層
  - 和系統有關的事件和命令。(e.g. 電池電量)
- 驅動層
  - 和網路卡驅動程式有關的事件和命令。(e.g. Link Up event)
- 網路層
  - 和網路層有關的事件和命令。 (e.g. IP Change Event)
- 傳輸層
  - 和傳輸層有關的事件和命令。(e.g. SET_TCP_RETX_TIMEOUT)

4.4.1 事件定義

以 NETDEV 開頭的事件，表示〔32〕中所支援的事件。

4.4.1.1 系統層事件

- NETDEV_REGISTER
  - 新的網路界面卡被加入到系統當中。
- NETDEV_UNREGISTER
  - 網路界面卡從系統中移除。
- NETDEV_CHANGENAME
  - 網路界面卡在系統內部的名稱改變了。

4.4.1.2 硬體驅動層事件

- NETDEV_UP
  - 網路界面卡被啓動了。
- NETDEV_DOWN
  - 網路界面被關閉了。
- NETDEV_REBOOT
  - 當網路介面偵測到硬體錯誤且需要重新啓動。
- NETDEV_CHANGEMTU
  - 網路介面的 MTU (Maximum Transfer Unit) 被更改。
- NETDEV_CHANGEADDR
  - 網路介面之硬體位址改變。
- LINK_UP
  - 網路界面卡可以開始傳送資料封包了。通常爲網路界面卡和接取點(POA)有了連接。關於各種不同類型的網路界面卡在何時會產生 Link UP 的事件，可參考〔37〕。
- LINK_DOWN
  - 某張網路界面卡無法傳送資料封包了。通常爲網路界面卡和接取點(POA)的連接斷掉了。關於各種不同類型的網路界面卡在何時會產生 Link DOWN 的事件，可參考〔37〕。
- POA_SCAN_COMPLETE
  - 描接取點 (POA) 的動作完成了，並且傳回接取點的資訊。這個事件通常發生在 POA_SCAN 的命令之後。
- SIGNAL_STRENGTH_THRESHOLD
  - 網路界面和所連接的 POA 訊號強度跨過程式設計者所指定的 THRESHOLD 時產生此事件通知程式設計者。

4.4.1.3 網路層事件

- IP_CHANGE
  - 網路界面卡上的網路位址 (IP Address) 改變了。
- DEFAULT_GATEWAY_CHANGE
  - 預設閘道改變的事件。當程式設計者同時註冊此事件和 ROUTING_TABLE_CHANGE 事件時，和預設閘道改變相關的，則只會收到 DEFAULT_GATEWAY_CHANGE 事件。
- ROUTING_TABLE_CHANGE
  - 路由表資料改變的事件。

4.4.1.4 傳輸層事件

- TCP_TX_STATE_CHANGE
  - TCP 傳送的狀態改變了。狀態一種爲 Slow Start，另一個爲 Congestion Avoidance。

4.4.2 命令定義

這邊的命令定義，就包括了控制和查詢界面。一般而言使用 SET 開頭的命令爲控制，使用 GET 開頭的命令爲查詢。因爲控制和查詢的界面都有對底層下達命令的動作，所以我們以命令的型態來到達兩者的目的。

4.4.2.1 系統層命令

- GET_ALL_NETWORK_INTERFACE
  - 查詢目前系統上的所有網路界面。

- GET_BATTERY_POWER
  - 查詢目前系統剩餘電力，適用於使用電池的裝置。

4.4.2.2 硬體驅動層命令

- SET_POA/ GET_POA
  - 爲網路界面卡設定接取點 (POA)。
  - 查詢網路界面卡目前接取點的資訊。
- POA_SCAN
  - 指定網路界面下達掃描鄰近接取點 (POA) 的命令。掃描結束後會由事件機制來通知。
- GET_SIGNAL_STRENGTH
  - 查詢網路界面卡和所連接的 POA 的訊號強度。
- SET_SIGNAL_THRESHOLD/GET SIGNAL_THRESHOLD
  - 設定和查詢訊號強度的門檻 (Threshold) 以百分比爲單位 100 爲最強，0 爲最弱 。當程式開發者不想要定期的去查詢目前網路界面所連結的POA的訊號強度時，可藉由設定signal threshold來讓WinME在訊號強度跨過指定的門檻時以事件的方式通知。爲了防止訊號強度波動會一直跨過門檻，WinME提供設定AVOID_PINGPONG_VALUE的命令。當 (目前訊號強度 > AVOID_PINGPONG_VALUE + SIGNAL_THRESHOLD) 時才會收到訊號強度大於門檻的事件。當 (目前訊號強度 < SIGNAL_THRESHOLD – AVOID_PINGPONG_VALUE) 時才會收到訊號強度小於門檻的事件。可參考 Figure 4-4。

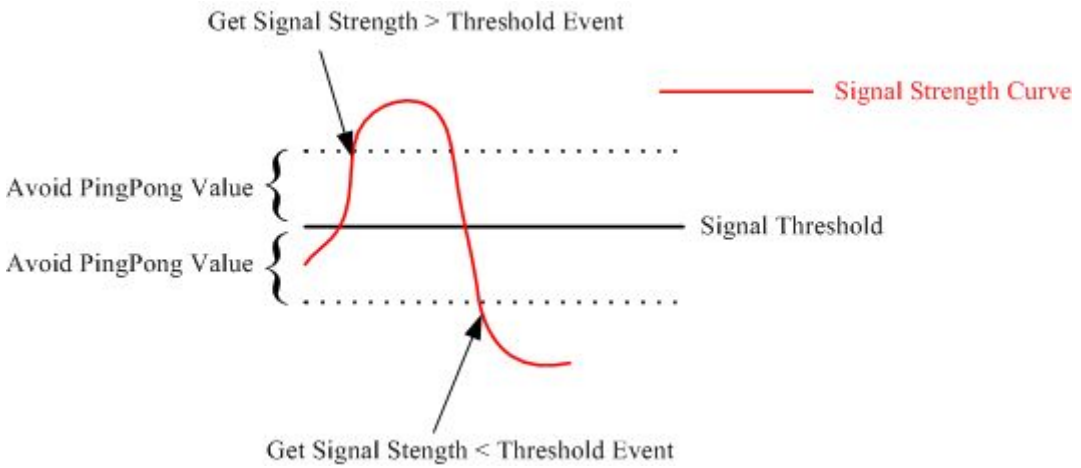


Figure 4-4 訊號強度和訊號門檻等關係圖

- SET_SIGNAL_AVOID_PINGPONG_VALUE
  - 當程式設計者有設定訊號門檻時，爲了防止訊號的波動一直產生訊號強度跨過訊號門檻的事件，可以設定防止乒乓值。
- GET_SIGNAL_AVOID_PINGPONG_VALUE
  - 查詢 AVOID_PINGPONG_VALUE 的值。
- SET_MTU/ GET_MTU
  - 設定和查詢網路界面卡的最大傳輸單位 (Maximum Transmission Unit, MTU)。Table 4-4 顯示一般常見網路界面的 MTU。

Table 4-4 一般常見網路界面的 MTU

網路界面	MTU (bytes)
EtherNet (一般的網路介面，這也是一般系統的預設值)	1500
PPPoE (ADSL 用的)	1492
Dial-up (modem)	576

- GET_MAC_ADDRESS
  - 查詢指定網路界面卡的 MAC Address。
- GET_BANDWIDTH
  - 查詢指定網路界面卡目前所連線網路的頻寬。目前沒有實作該命令
- GET_STATISTICS
  - 查詢網路界面卡的統計值。如收到封包總數、傳送封包總數等…。

4.4.2.3 網路層命令

- SET_STATIC_IP
  - 設定網路界面的網路位址 (IP ADDRESS)。
- SET_IP_BY_DHCP
  - 使用 DHCP 爲網路界面卡取得網路位址。
- GET_IP
  - 查詢網路界面卡上目前所設定的網路位址。
- SET_NETMASK/ GET_NETMASK
  - 設定或查詢網路界面卡的子網路遮罩。
- SET_BROADCAST/GET_BROADCAST
  - 設定廣播位址 (Broadcast) 的值。Broadcast 的算法會按照 IP 位址與 Netmask 的不同，會有不同的值。一般來說，未設定 broadcast，但 IP 位址與 Netmask 設定正確的話，網路還是可以正常動作，所以習慣上，常不會做 Broadcast 的設定。
- SET_DEFAULT_GATEWAY/GET_DEFAULT_GATEWAY
  - 設定預設閘道，這也可以使用 SET_ROUTING_TABLE 的命令達成，不過預設閘道算是一個比較特別的路由表資料，所以獨立出一個命令。
- SET_ ROUTING_TABLE/GSET_ ROUTING_TABLE
  - 設定和查詢路由表資料。

4.4.2.4 傳輸層命令

- REGISTER_TCP_TX_STATE_EVENT
  - 當 TCP 的傳送狀態改變時通知。這裡的狀態不是 TCP 狀態機。而是 TCP 在避免擁塞時所處的狀態，只有兩種狀態分別爲 Slow Start 或 Congestion Avoidance。

- SET_TCP_TX_STATE /GET_TCP_TX_STATE
  - 設定和查詢 TCP 連線目前的傳送狀態。
- SET_TCP_RETX_TIMEOUT/GET_TCP_RETX_TIMEOUT
  - 設定和查詢目前 TCP 連連沒收到 Ack 就重送封包的 Timeout 值。
- SET_TCP_RTT/GET_TCP_RTT
  - 設定和查詢目前 TCP 連線在系統核心中的 Round Trip Time (RTT) 值
- SET_TCP_RECV_WIN/GET_TCP_RECV_WIN
  - 設定和查詢目前 TCP 連線在核心內的 Receive Window 大小。
- SET_TCP_CONG_WIN /GET_TCP_CONG_WIN
  - 設定和查詢目前 TCP 連線在核心內的 Congestion Window 大小。

4.5 應用程式界面 (API) 設計

本節將會介紹本論文所提之 WinME 的應用程式界面。值得注意的是在本論文的設計中應用程式和 WinME 都是透過訊息在溝通的，不論是哪個應用程式介面，都只是傳遞訊息給 WinME 而已。而 WinME 會跟據收到的訊息執行相關的動作。

4.5.1 傳送事件資訊的資料結構 win_event structure

當事件產生的時候，程式開發人員不會只想知道事件的發生，他們一定還需要其它資訊，比如說一個Link UP的事件產生了，程式人員還會想知道是哪張網路界面卡產生此事件，或著說一個 IP 位址改變的事件發生了，程式設計人員還需要知道 IP 位址變爲多少。所以事件的產生是伴隨著相關資訊的。所以我們定了一個用來傳達事件資訊的資料結構叫作 win_event。Table 4-5 顯示了 win_event 資料結構的定義。我們採取一個通用的資料結構來傳遞事件資料是爲了簡化API的設計，有了一個通用的資料結構，我們就可以只使用一個以該資料結構爲參數的函式來處理我們所定義的所有事件。

Table 4-5 win_event 資料結構的定義

```
struct win_event {
      INT32      event;
      VOID*      value;
};
```

win_event 的第一個欄位就是用來告知所產生的網路事件，第二個欄位就是該網路事件的相關資訊。我們可以看到 value 的欄位爲 VOID* 型別，這是因爲每個網路事件所伴隨的資訊都不一定會完全一樣，有的是 IP 位址、有的是網路界面卡 ID、有的是訊號強度等，因此我們不預先決定 value 的型別，value 就可以在應用程式收到事件後再去動態的解釋就行了。我們在命令相關的應用程式界面上也是採取類似的作法。

Figure 4-5 顯示了NETDEV_REGISTER事件所伴隨的訊息格式，我們可以看到第 1 個BYTE是網路卡的ID，之後 9 個bytes爲網路卡在系統內部的名稱，所以很明顯地我們就可以知道是哪張網路界面卡被新增到系統當中。

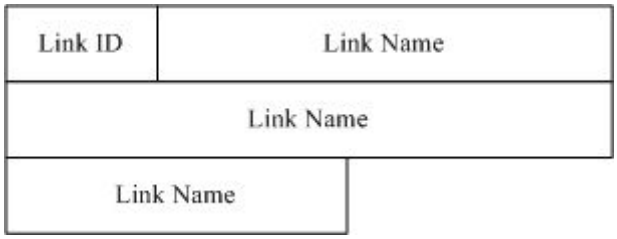


Figure 4-5 NETDEV_REGISTER 事件的 value 格式

再來看一個IP_CHANGE 的例子。Figure 4-6 顯示了當 IP_CHANGE事件產生時所伴隨的資訊。我們可以看到第 1 個byte爲網路界面卡ID，用於告訴應用程式是哪張網路界面卡的IP位址改變了。後面就是一般常見的IP位址相關資訊。

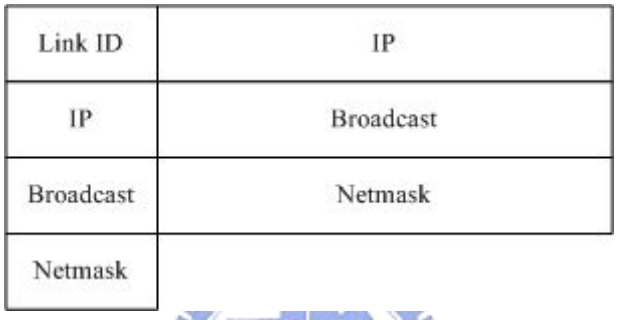


Figure 4-6 IP_CHANGE 事件的 value 格式

4.5.2 事件初始化的程式界面 win_init( )

前面 4.3 有提到事件在使用的時候需要先作初始化的動作，讓WinME配置事件佇列，和決定要同步處理事件或非同步處理事件。初始化完成後會傳回事件描述子給應用程式。這個初始化的程式界面就爲 win_event_init。Table 4-6 顯示了 win_event_init 的原型 (Prototype)。

Table 4-6 win_event_init 的原型

```
INT32 win_event_init(
      VOID (*event_handler)(struct win_event*)
);
```

我們可以看到 win_event_init 的參數是一個函式指標，當參數爲 NULL 時，表示應用程式想要採用同步處理的方式來自己檢查事件。如果應用程式有提供自己的 event_handler 的話，就表示事件的處理是交給執行緒來處理，該執行緒等到事件發生時就會執行應用程式所提供的 event_handler 來處理事件。由於是執行緒來處理事件，所以原本的應用程式不論執行到哪段程式碼，當事件發生時都還是可以被執行緒所處理，所以稱爲不同步的事件處理方式。我們也看到 event_handler 的參數爲 win_event，所以當事件發生時，事件的相關資訊都會由參數傳遞進來。win_event_init 傳回值就是事件描述子，應用程式可以藉著這個事件描述子來註冊感興趣的事件或檢查是否有事件產生。

4.5.3 註冊感興趣事件的程式界面 win_event_register( )

當事件初始化之後，應用程式需要註冊感興趣的事件，因爲很少會有應用程式會想要所有的網路事件，它們只想要它們需要的。所以我們提供了 win_event_register 這個註冊事件的應用程式界面。讓應用程式可以只收到它們感興趣的事件。Table 4 7 顯示了 win_event_register 的原型 (Prototype)。

Table 4-7 win_event_register 的原型

```
INT32 win_event_register(
      INT32 event_descriptor,
      INT32 event
);
```

第一個參數爲事件描述子，這是在初始化 (win_event_init) 時得到的。第二個參數就是要註冊的事件。只有用win_event_register註冊過的事件才會傳給事件描述子對應的事件佇列。win_event_register 的傳回值如 Table 4-8 所示。

Table 4-8 win_event_register 的傳回值

R_OK	執行成功
R_FAIL	因爲某種原因而執行失敗
R_INV_PARAM	不合法的參數。

R_NOT_SUPPORT	此中介軟體不支援這個事件。

4.5.4 檢查事件發生的程式界面 win_check_event( )

在同步的事件處理方式下，應用程式需要檢查是否有事件的產生。所以我們也提供了檢查事件產生的程式界面win_check_event。Table 4-9 顯示 win_check_event 的原型 (Prototype)。

Table 4-9 win_check_event 原型

```
INT32 win_check_event(
    INT32              event_descriptor,
    struct win_event*  t_event,
    BOOL               block
);
```

event_descriptor爲事件描述子，之前已經介紹過每個事件描述子會對應一個事件佇列，win_check_event就是去檢查該事件佇列是否有事件。如果有事件發生的話，關於該事件的資訊會存放在t_event的資料結構中。參數block指定要不要等到事件產生。參數block如果爲FALSE，則沒有事件的話會馬上返回，參數block 如果爲TRUE，則應用程式會等到有事件時才會返回。win_check_event的傳回值如 Table 4-10 所示。

Table 4-10 win_check_event 的傳回值

R_HAVE_EVENT	有事件產生。
R_NO_EVENT	沒有事件產生。

4.5.5 包裝控制及查詢動作的命令資料結構 win_cmd structure

由於不想針對每個不同的命令都設計一個程式界面，如 set_ip( )、set_mtu( )、get_signal_strength( )等…，所以我們將不同的命令和參數都填入 win_cmd的資料結構，然後再傳入win_do_cmd的程式界面當參數。這樣可以簡化程式界面的設計，而且也具有擴充性，如果以後有新的命令需要支援，我們只要定義新的命令和對應的參數就行了，不需要產生新的應用程式界面。Table 4-11 顯示了 win_cmd 資料結構的定義。

Table 4-11 win_cmd 的資料結構定義

```
struct win_cmd {
      INT32       cmd;
      VOID*     param;
      VOID*     value;
};
```

cmd 欄位指定了應用程式想要執行的命令。

param 欄位表示該命令的參數，如果該命令需要參數的話就是填在這個欄位。

value 欄位通常是給查詢命令使用的，當命令是爲查詢網路資訊時，則查詢到的資訊就會被存放在 value 的欄位傳回給應用程式。還有一種情況會用到此欄位，就是當命令執行出現問題的時候，中介軟體會將錯誤訊息用字串的方式存放在此欄位後回報。

param 和 value 欄位和 win_event 的 value 欄位一樣都是 VOID* 的型別，這表示 param 及 value 欄位也不是事先就定好的，它們也是根據不同的命令有不同的解釋方法。以下會舉幾個例子。

當應用程式想要知道目前系統上有哪些網路界面卡，則可以用 GET_ALL_NETWORK_INTERFACE的命令，這個命令是不用填param參數欄位的。傳回的資訊則會存放在 value欄位裡，格式如 Figure 4-7 所示。第一個byte指出系統上有多少個網路界面卡，之後每 11 bytes都分別爲一個網路界面卡的資訊。有網路界面卡ID、網路界面卡型態、和網路界面卡在系統內的名字。我們也可以發現此時 value的大小是變動的，不是固定大小，它的大小則取決於 Interface Num，也就是系統內有多少張網路界面卡。

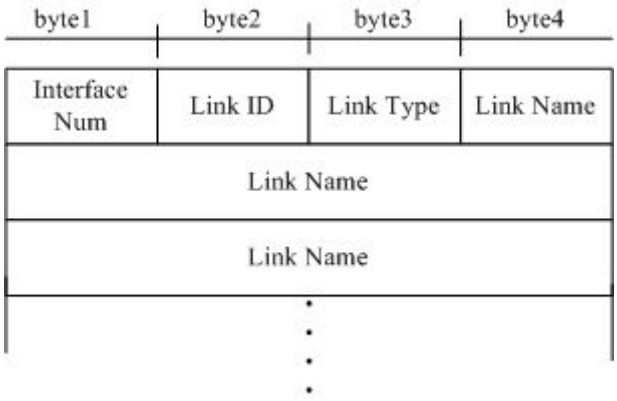


Figure 4-7 GET_ALL_NETWORK_INTERFACE 的 value 格式

應用程式想要幫網路界面卡設定IP位址時，則使用SET_STATIC_IP的命令，直覺地這個命令一定需要參數來指定要設的網路界面卡及IP位址，所以SET_STATIC_IP命令的param格式就如 Figure 4-8 所示。第一個 byte 表示網路界面卡的ID，用來指定要設定IP位址的網路界面卡，後面有兩個 4 byte的分別表示要設定的IP位址和子網路遮罩。所以在這個命令下value欄位是用不到的。

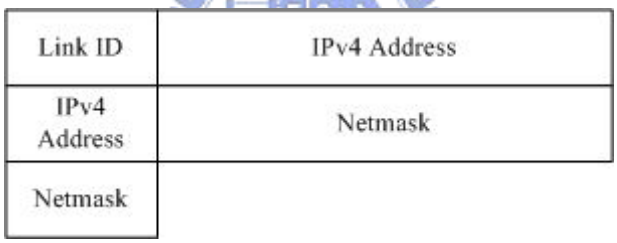


Figure 4-8 SET_STATIC_IP 的 param 格式

4.5.6 執行命令的程式界面 win_do_cmd( )

由於我們提出了 win_cmd 的資料結構，所以執行命令的應用程式界面只需有一個 win_do_cmd 就行了。當應用程式將 win_cmd 資料結構填好要執行的命令和參數後，只要將該 win_cmd 資料結構當參數傳進 win_do_cmd，WinME 就會去執行該命令。Table 4 12 顯示了 win_do_cmd 的原型。這邊有一件事要注意的是，win_do_cmd 並不會直接執行應用程式下達的命令，它只有把要執行命令的訊息傳遞給 WinME，然後再由 WinME 自行找機會執行。

Table 4-12 wn_do_cmd 的原型 (prototype)

```
INT32 win_do_cmd (
      struct win_cmd* cmd
);
```

我們可以看到win_do_cmd非常的簡潔，只有一個 win_cmd資料結構的參數，這是因爲我們將複雜的事物都由win_cmd封裝起來處理掉了。也因爲如此，我們的中介軟體就不需提供許許多多不同的應用程式界面來處理許多不同的命令。Table 4-13 顯示了 win_do_cmd 的傳回值。

Table 4-13 win_do_cmd 傳回值

R_OK	命令執行成功
R_FAIL	命令因爲某種原因而執行失敗。失敗原因爲以字串型式存放在win_cmd結構中的pv_value欄位可參考 Table 4-11。
R_INV_PARAM	不合法的參數。
R_NOT_SUPPORT	此中介軟體不支援這個命令。

第 五 章 WinME 在 Linux 作業系統下的實作

5.1 軟硬體需求

以下是本論文實作之軟硬體平台：

- 硬體需求：IA-32 Intel architecture compliant computer
- 系統核心：Linux kernel version 2.4.32
- 發行套件：Red Hat Linux Release 9
- 發展工具：
  - GNU gcc 3.2.2 (Red Hat Linux 3.2.2-5)
  - GNU ld 2.13.90
  - GNU make 3.79.1

5.2 查詢和控制命令的實作

命令的實作部份是比較簡單的，原則上來講就是應用程式透過 WinME 去跟系統底層下達命令。而處理系統底層的程式碼是由 WinME 作掉，應用程式只需查看 WinME 放出來的命令來使用即可。

當應用程式填好 win_cmd 資料結構後呼叫 win_do_cmd 函式時，win_do_cmd 的動作只有把該資料結構的命令和參數欄位包裝成一個訊息封包後傳送給 WinME，然後 win_do_cmd 函式就會等待 WinME 的回傳資訊，等到 WinME 把資訊回傳之後，再將該訊息的資料取出，填入 win_cmd 資料結構的 value 欄位，然後再返回呼叫。

因爲可能同時會有多個應用程式呼叫 win_do_cmd 函式，所以 WinME 並不是馬上處理應用程式們所有要求的命令，而是把一些來不及處理的命令訊息先放到命令佇列中，等到前一個命令處理完後再處理下一個。這邊要再強調一下，當應用程式呼叫 win_do_cmd 時並不是直接去執行系統底層的程式碼，而是把要執行系統相關動作的要求傳送給 WinME，請 WinME 幫忙執行。舉個例來說，當行動管理員程式使用 win_do_cmd 來更改 IP 位址時，更改 IP 位址的動作並不是在行動管理員的程序下執行的，而是在 WinME 的程序下執行的。

我們不使用類似函式庫的作法，也就是將底層的程式碼實作在 win_do_cmd 中，而是將所要執行的命令以訊息的方式傳送給 WinME 中央控管的好處有：

- WinME 可以控制資源的使用
- WinME 可以幫命令排優先權
- 只有 WinME 處理系統相關動作，win_do_cmd 的移殖可變的容易

缺點爲就是當有很多命令在佇列中等待時，應用程式就需要等較長的時間。Table 5-1 顯示了 win_do_cmd 的虛擬碼，我們可以發現在win_do_cmd中沒有任何關於系統底層的動作。

Table 5-1 win_do_cmd 的虛擬碼

```
INT32 win_do_cmd (struct win_cmd* cmd) {
      pack cmd;
      send to WinME;
      wiat for WinME's response;
      fill info into cmd.value field;
      return;
}
```

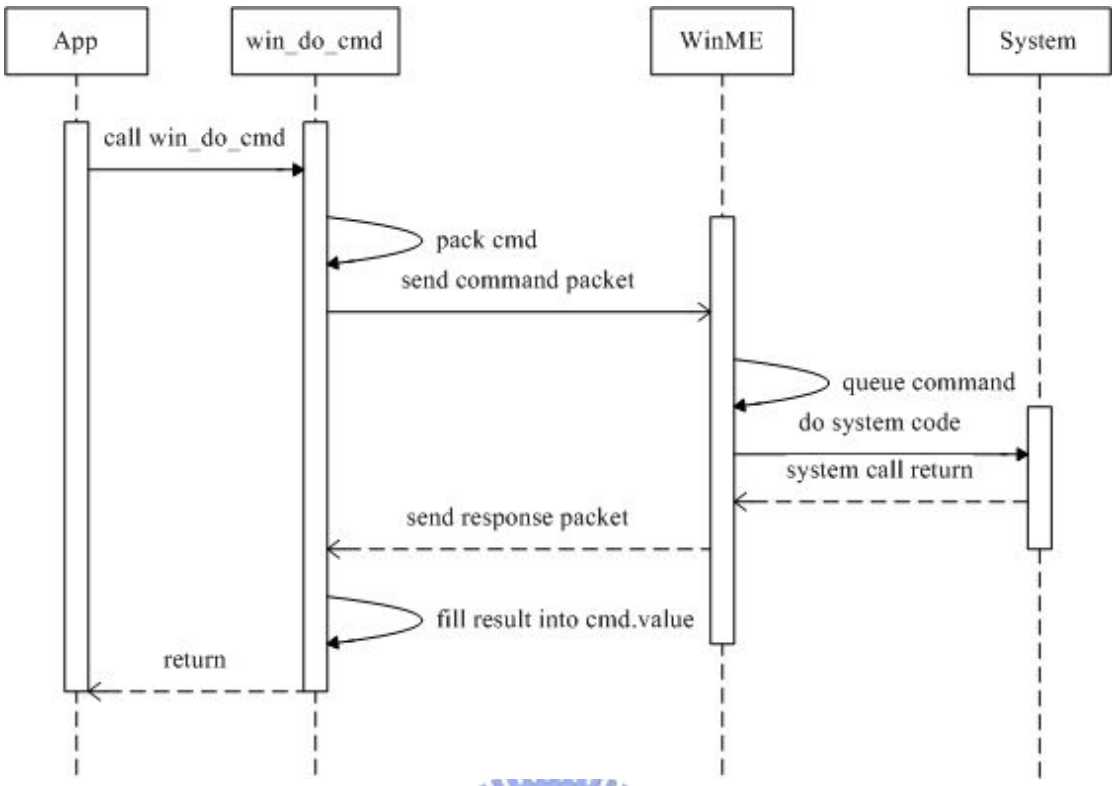


Figure 5-1 win_do_cmd 和中介軟體等序列圖

Figure 5-1 顯示了應用程式呼叫 win_do_cmd和中介軟體等序列圖。Table 5 2 則顯示了各步驟的說明。

Table 5-2 Figure 5-1 的各步驟說明

步驟	描述
call win_do_cmd	應用程式包裝好 win_cmd 資料結構後，呼叫 win_do_cmd，並且將 win_cmd 當作參數傳入。
pack cmd	win_do_cmd 先將 win_cmd 資料結構中關於命令和參數的欄位封裝成一個命令訊息封包。
send cmd packet	將前一個步驟所產生的命令訊息封包傳送給中介軟體。
queue command	中介軟體將訊息封包放入命令佇列中。
do system code	執行命令佇列裡的應用程式所請求的命令訊息，這時會呼到到系統函式。
system call return	系統函式執行成功後回傳。
send response packet	中介軟體將執行結果包成封包後傳回給 win_do_cmd
fill result into cmd.value	win_do_cmd 函式將執行結果放入 win_cmd 資料結構的 value 欄位。
return	返回呼叫。

5.3 事件通知機制的實作

這一節會分兩個部份介紹，第一個部分是WinME核心的事件服務元件 (Event Service) 的實作，第二部分爲系統底層的事件如何傳達給事件服務元件。

5.3.1 事件服務元件

應用程式透過 win_event_init、win_event_register、win_check_event 和 WinME 的事件服務元件溝通。本節就是介紹這一部分在 Linux 系統下的實作方法。

在 Linux 下已經有一個存在的機制 socket，跟我們的網路事件通知機制的設計很符合，所以我們在 Linux 系統下事件服務元件和應用程式的溝通就採用目前已存在的機制 socket 完成的。當應用程式呼叫 win_event_init 的時候，事實上就是透過 socket 和事件服務元件建立一個 Streaming 的連線，而傳回的事件描述子就是建立連線後的 socket file descriptor。

當應用程式呼叫完 win_event_init 後，WinME 這邊的事件元件也會有一個對應的 socket file descriptor，所以只要網路底層的事件產生的話，就可以直接使用該 socket file descriptor 把網路事件傳達給應用程式。

那如何知道哪些事件才是應用程式感興趣的事件呢？我們在事件服務元件中，每個應用程式所對應的 socket file descriptor 都會有一個對應的表格，該表格就記錄著應用程式感興趣的事件。於是當網路事件發生時，事件服務元件會去掃描所有註冊過的 socket file descriptor 所對應的表格，發現有註冊過該事件，就將送出該網路事件的資訊。

Figure 5-2 顯示了我們的實作方式。所以在我們的事件服務元件中是不曉得爲哪個應用程式需要事件，事件服務元件只認得連線的socket file descriptor。可以將 Figure 5-2 和 Figure 4-2 作比較。

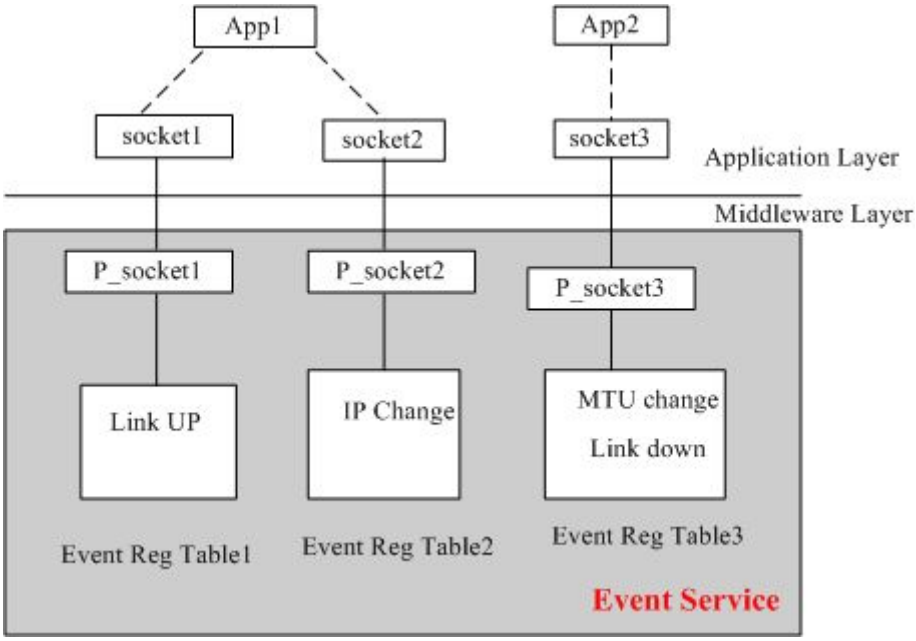


Figure 5-2 Event Service 和應用程式通訊的實作方式

要使用 win_event_register 註冊事件時也是透過 socket 傳達註冊事件的訊息給事件服務元件。事件服務元件就會把對應的資料加到事件註冊的表格內。

我們知道事件服務元件和應用程式間的溝通是用socket 之後，就可以很簡單的實作win_check_event這個函式了。我們直接使用select函式來檢查是否有訊息可以讀取就行了。當發現有訊息可以讀取時，就將該訊息讀進win_event的資料結構中後返回。沒有訊息可以讀取的時候，就看應用程式傳進來的block參數，如果是FALSE的話，我們就讓select函式等待 0 秒，如果是TRUE的話，就讓select函式等待到有訊息可讀取。Table 5-3 顯示了 win_check_event 的實作程式碼。

Table 5-3 win_check_event 在 Linux 下實作程式碼

```
INT32 win_check_event(
                INT32       event_descriptor,
                struct      win_event* t_event,
                INT32       size,
                BOOL        block
          )
{
   <--    some initial code -->
       if(block) {
             nready = select(event_descriptor + 1, &rset, NULL, NULL, NULL);
       }
       else {
             nready = select(event_descriptor + 1, &rset, NULL, NULL, &timeout);
       }
       if(nready == 0) {
             return R_NO_EVENT;
       }
       <-- fill t_event structure -->
       return R_HAVE_EVENT;
}
```

5.3.2 系統底層的網路事件

在 Linux 系統下已經有一個機制可以將核心層的訊息傳送到使用者空間的應用程式中，這個機制就是 Netlink〔29〕。而建構在Netlink上的rtnetlink更可以將核心內部的網路事件以訊息的方式傳送給使用者空間的應用程式。由於在Linux下已經有一個機制可以讓應用程式取得網路事件，爲了避免造新的輪子，因此我們決定採用rtnetlink來將核心層的網路事件傳送到事件服務元件。不過原本的rtnetlink socket對於我們的WinME而言還是有以下幾點缺失：

- 支援事件的數量不足。
  - 對於本論文的中介軟體而言，rtnetlink 有許多事件尚未支援。如 Link UP。

- 當事件發生時，所提供的資訊無法分類是何種事件發生。
  - e.g. 當網路卡的狀態改變時不論是 MTU 改變，或著是網卡被啓動了，rtnetlink 都一律只提供 RTM_NEWLINK 型態的事件。如果我們想要知道是何種事件的發生，就需要把所有的屬性值都記錄下來，然後和之前的作比較，才可以知道是哪個屬性的值更動了，如要知道是 MTU 改變的事件，就比較新的 MTU 值和舊的 MTU 值有無改變，這樣顯得就沒有效率。所以我們就修改 rtnetlink 在核心中實作的方式使之可以在通知事件產生時一併通知是何種事件的發生。

由於rtnetlink對我們的WinME而言有以上兩點缺失，所以我們會去擴充核心中原本rtnetlink的實作方法，除了增加新的觸發點[1]外，還增加了一些屬性(Attribute)。觸發點用於提供原本netlink未提供的事件，新的屬性可以幫助我們分辨事件。

5.3.3 網路層事件

本論文所提之中介軟體中目前只定義了三個網路事件，分別爲 IP_CHANGE、DEFAULT_GATEWAY_CHANGE 和 ROUTING_TABLE_CHANGE，而這三個事件在 rtnetlink 中已經有支援了，所以這部份的實作就我們就直接將從 rtnetlink 得的訊息資訊重新包裝成本論文所提中介軟體的所定的訊息格式即可。

當 rtnetlink 傳來的訊息型態爲 RTM_NEWADDR 時，就可以知道網路界面卡的 IP 位址改變了，此時就可以將改變後的 IP 位址從 IFA_ADDRESS 的屬性中讀取出來，

[1]此處的觸發點是指核心中送出 rtnetlink 訊息給使用者空間應用程式的點。

當 rtnetlink 傳來的訊息型態爲 RTM_NEWROUTE 或著是 RTM_DELROUTE 時，我們就可以知道是路由表的資訊改變了，NEW 表示有資料新增進來了，DEL 表示有資料被刪除。而目的地、閘道及出去的網路界卡資料分別爲屬性 RTA_DST、RTA_GATEWAY、RTA_OIF。如果要知道是否爲預設閘道改變的資訊，我們就直接看 RTA_DST 屬性的長度是否爲 0 就知道了。

5.3.4 NETDEV_XXX 相關事件

在 Linux 的核心中已經提供NETDEV_XXX的訊息通知機制，就是使用 notifier chain〔38〕。由於 rtnetlink在核心中實作時已經有註冊一個 notifier block，所以我們也毋需註冊新的notifier block，而直接修改核心中rtnetlink實作的程式碼。由於原本的實作所提供的資訊太少了，不論是何種的NETDEV事件，原本的rtnetlink全部歸類爲RTM_NEWLINK或RTM_DELLINK的型態。

Table 5-4 顯示了rtnetlink在核心中處理NETDEV_XXX事件的程式碼，我們可以發現它所提供的資訊傳到使用者空間的應用程式時，關於NETDEV_XXX的訊息完全不見了，而且有很多NETDEV_XXX的事件都沒經過特別處理，一律規類在default中處理，而且處理方式就只是將網路界面卡的資料傳上使用者空間，然後給個型態 RTM_NEWLINK而已。

因此我們擴充了這一部份的程式碼，爲了直接分辨系統層事件或著是驅動層事件，所以我們增加了兩個訊息型態分別爲 RTM_SYSTEM_EVENT 及 RTM_DRIVER_EVENT，然後又增加了一個屬性爲 IFLA_WIN_EVENT 用來指名是何種 NETDEV_XXX 的事件發生。爲了保持原本 rtnetlink 在這邊的運作，我們會自己再送一個 rtnetlink 的訊息出來，然後裡面填的資料就是 WinME 想要知道的資料。因此當 notifier chain 呼叫到 rtnetlink 的 notifier block 時，是會送出兩個 rtnetlink 的訊息的，一個是原本的，一個是我們所新增的。

Table 5-4 rtnetlink 在核心中 notifier block 的部分程式碼

```
        case NETDEV_UNREGISTER:
                rtmsg_ifinfo(RTM_DELLINK, dev, ~0U);
                break;
        case NETDEV_REGISTER:
                rtmsg_ifinfo(RTM_NEWLINK, dev, ~0U);
                break;
        case NETDEV_UP:
        case NETDEV_DOWN:
                rtmsg_ifinfo(RTM_NEWLINK, dev, IFF_UP|IFF_RUNNING);
                break;
        case NETDEV_CHANGE:
        case NETDEV_GOING_DOWN:
                break;
        default:
                rtmsg_ifinfo(RTM_NEWLINK, dev, 0);
                break;
        }
```

Table 5-5 修改過的 rtnetlink 核心程式碼

```
    case NETDEV_UNREGISTER:
        rtmsg_ifinfo(RTM_DELLINK, dev, ~0U);
        win_rtmsg_ifinfo(event, RTM_SYSTEM_EVENT, dev, ~0U);
        break;
    case NETDEV_REGISTER:
        rtmsg_ifinfo(RTM_NEWLINK, dev, ~0U);
        win_rtmsg_ifinfo(event, RTM_SYSTEM_EVENT, dev, ~0U);
        break;
```

5.3.5 其它的事件

關於其它事件，在核心中的實作方式就是在改變狀態的核心程式碼中增加一個 rtnetlink 的觸發點即可。

可是有一點要注意的，就是當改變狀態的核心程式碼是在中斷環境 (Interrupt Context) 下處理時，我們是沒有辦法直接傳送 rtnetlink 的訊息給使用者空間的，因爲在中斷環境下是不能處理會進入睡眠模式的或是進行 block I/O 動作的程式碼的。可是通常和驅動程式有關的狀態大部分都是在中斷環境中所改變的，因爲當硬體上的資訊改變時通常都是藉由中斷來通知系統。所以這個時候就沒辦法靠增加 rtnetlink 的觸發點來傳送訊息。

要解決以上的問題，Linux 在 2.4 版的核心中有提供 task queue 的下半部 (BottomHalf) 處理方式，而其中的 scheduler queue 中的 task 則可以在程序環境 (Process Context) 下所執行的。

所以我們在中斷環境下新增一個名稱爲 win_task_queue 的 scheduler queue，把我們要執行的傳送 rtnetlink 訊息給使用者空間的函式加入 win_task_queue 中，然後呼叫核心中提供的 schedule_task 函式將 win_task_queue 爲參數傳入，這樣就可以在程序環境中執行我們的函式。schedule_task 會喚醒核心執行緒 keventd 來處理 win_task_queue 中所佇列的函式。

以 LINK_UP和LINK_DOWN的事件爲例，當網路卡驅動程式發現這是一個LINK_UP或是LINK_DOWN的中斷時，驅動程式會去執行netif_carrier_on來將核心中維護網路卡狀態的carry bit 開啓，或是執行 netif_carrier_off 將 carry bit 清掉，不過這些都是在中斷環境中所執行的，因此我們沒辦法直接在netif_carrier_xx的函式中傳送LINK_UP或LINK_DOWN的訊息給使用者空間。所以這邊我們就以task queue的方式解決。相關程式碼可參考 Table 5-6。

Table 5-6 送 LINK_UP、LINK_DOWN 事件的核心程式碼

```
/* use rtnetlink socket to notify user space LINK events */
static void win_notifier_taskq(void* dummy)
{
      /* if nocarry bit is zero, send NETDEV_LINKDOWN event */
      if(test_bit(__LINK_STATE_NOCARRIER, &win_dev->state)) {
            win_rtmsg_ifinfo(NETDEV_LINKDOWN, RTM_DRIVER_EVENT,…);
      }
      else {
            win_rtmsg_ifinfo(NETDEV_LINKUP, RTM_DRIVER_EVENT, ..);
      }
}
/* declare a scheduler queue */
static struct tq_struct linkwatch_queue;

static int linkwatch_fire_event(struct net_device *dev)
{
      <-- some initial code -->
      /* call schedule_task to process functions in linkwatch_queue */
      schedule_task(&linkwatch_queue);
}
/* when interface card connect POA, it's driver will call this routine*/
void netif_carrier_on(struct net_device *dev)
{
      if (test_and_clear_bit(__LINK_STATE_NOCARRIER, &dev->state))
            linkwatch_fire_event(dev);
      if (netif_running(dev))
            __netdev_watchdog_up(dev);
}
/* when interface card disconnect POA, it's driver will call this routine*/
void netif_carrier_off(struct net_device *dev)
{
      if (!test_and_set_bit(__LINK_STATE_NOCARRIER, &dev->state))
            linkwatch_fire_event(dev);
}
```

第 六 章 成果及貢獻

我們所發展出的 WinME，對於程式開發人員開發具網路感知的應用程式有很大的幫助，藉著事件通知機制，應用程式毋需在系統內發出系統呼叫來輪詢底層的網路狀態，而是在網路狀態改變時可以馬上被告知網路各層狀態的變化，進而可以立即調整本身的行爲。

而所提供的命令界面可以讓程式開發人員不需要撰寫系統底層相關的程式碼就可以操作系統內部協定推疊各層的狀態，減少寫程式出錯的機率和撰寫底層相關程式碼的繁瑣工作。Table 6 1 顯示了在 Linux 下，程式開發人員如何將目的位址爲 192.168.0.100、經過系統內編號爲 2 的網路界面卡的路由表資料加到路由表中。而 Table 6 2 則是利用 WinME 所提供的程式界面作到一樣的事情，我們可以很明顯地發現程式碼簡潔了不少。

Table 6-1 Linux 下使用 rtnetlink socket 新增一個資料到路由表中的程式碼片段

```
int main(int argc, char *argv[])
{
   ....
   /* attributes of the route entry    */
   char dsts[24] = "192.168.0.100";
   int ifcn = 2, pn = 32;
   /* open socket */
   fd = socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE);
   /* setup local address & bind using this address */
   bzero(&la, sizeof(la));
   la.nl_family = AF_NETLINK;
   la.nl_pid = getpid();
   bind(fd, (struct sockaddr*) &la, sizeof(la));
   /* initialize RTNETLINK request buffer */
   bzero(&req, sizeof(req));
   /* compute the initial length of the service request */
   rtl = sizeof(struct rtmsg);
```

```
/* add first attrib: */
/* set destination IP addr and increment the RTNETLINK buffer size*/
rtap = (struct rtattr *) req.buf;
rtap->rta_type = RTA_DST;
rtap->rta_len = sizeof(struct rtattr) + 4;
inet_pton(AF_INET, dsts, ((char *)rtap) + sizeof(struct rtattr));
rtl += rtap->rta_len;
/* add second attrib: */
/* set ifc index and increment the size */
rtap = (struct rtattr *) (((char *)rtap)    + rtap->rta_len);
rtap->rta_type = RTA_OIF;
rtap->rta_len = sizeof(struct rtattr) + 4;
memcpy(((char *)rtap) + sizeof(struct rtattr), &ifcn, 4);
rtl += rtap->rta_len;
/* setup the NETLINK header */
req.nl.nlmsg_len = NLMSG_LENGTH(rtl);
req.nl.nlmsg_flags = NLM_F_REQUEST | NLM_F_CREATE;
req.nl.nlmsg_type = RTM_NEWROUTE;
/* setup the service header (struct rtmsg) */
req.rt.rtm_family = AF_INET;
req.rt.rtm_table = RT_TABLE_MAIN;
req.rt.rtm_protocol = RTPROT_STATIC;
req.rt.rtm_scope = RT_SCOPE_UNIVERSE;
req.rt.rtm_type = RTN_UNICAST;
/* set the network prefix size */
req.rt.rtm_dst_len = pn;
/* create the remote address to communicate */
bzero(&pa, sizeof(pa));
pa.nl_family = AF_NETLINK;
/* initialize & create the struct msghdr supplied to the sendmsg() function */
bzero(&msg, sizeof(msg));
msg.msg_name = (void *) &pa;
msg.msg_namelen = sizeof(pa);
/* place the pointer & size of the RTNETLINK    message in the struct msghdr */
iov.iov_base = (void *) &req.nl;
iov.iov_len = req.nl.nlmsg_len;
msg.msg_iov = &iov;
msg.msg_iovlen = 1;
```

```
    /* send the RTNETLINK message to kernel */
    rtn = sendmsg(fd, &msg, 0);
    /* close socket */
    close(fd);
}
```

Table 6-2 使用 WinME 所提供的程式界面新增一個資料到路由表中的程式碼片段

```
struct set_routing_table_param {
        UINT8       act;
        UINT32      dst;
        UINT32      gateway;
        UINT32      netmask;
        UINT8       link_id;
};
int main(int argc, char* argv[])
{
   /* routing table's attribute, dst means destination address
     * ifcn = 2 means out network interface's Link ID is 2
     * netmask is 255.255.255.255 means destination is a host    */
       char dsts[24] = "192.168.0.100";
       char netmask[24] = "255.255.255.255";
       int ifcn = 2;
       struct win_cmd cmd;
       struct set_routing_table_param *srtp;

       cmd.i8_cmd = SET_ ROUTING_TABLE;
       cmd.pv_param = (VOID*)malloc(sizeof(struct set_routing_table_param));
       srtp = (struct set_routing_table_param *)cmd.pv_param;
       /* fill param field */
       srtp->act = 1;
       inet_pton(AF_INET, dsts, ((char *)srtp->dst);
       srtp->gateway = 0;
       inet_pton(AF_INET, netmask, ((char *)srtp->netmask);
       srtp->link_id = ifcn;
       /* call win_do_cmd to send command message to WinME */
       win_do_cmd(&cmd);
       free(cmd.pv_param);

       return 0;
}
```

6.1 與其它系統比較

802.21〔30〕目前還在發展階段，和本論文提出系統比較如下：

- 802.21 是正在發展中的標準，而它需要底層的標準 (e.g. 802.16, 802.11) 作適度的修改來配合。
- MIH Function 和本論文所提的中介軟體類似，不過 MIH Function 只著重在處理 Layer2 以下的資訊。而本論文所提出的中介軟體是位於使用者空間的應用程式再藉著跨層的機制將 Layer2、Layer3 的資訊傳送上來，所以本論文所提的中介軟體不止可以處理 Layer2 的資訊，也可以處理 Layer3 之上應用層之下的資訊。

Gollum〔31〕和本論文所提系統比較如下：

- Gollum 目前只有提供一個通用的程式界面讓應用程式來設定和存取 Link Layer 層。所以無法操作協定堆疊內的其它層。而本論文所提出的系統有提供界面讓應用程式調整協定堆疊中其它層 (e.g. Network Layer or Transport Layer) 的行爲。

本論文和Linux平台上驅動程式層網路事件通知機制〔32〕的比較如下：

- 事件通知和處理的方式
  - ■ 該論文藉著從核心返回使用者空間時去檢查網路事件的產生，然後再使用信號機制來處理網路事件，這過程中還會去更改了使用者程式的核心堆疊，可能會對核心產生無法預期的影響。
  - ■ 本論文處理網路事件的方式是在使用者空間中處理的，程式可以自行檢查事件的產生，或著是使用另一個執行緒 (thread) 來處理事件，其中完全不會更動核心中的任何程式碼。
- 本論文除了提供該論文的所有事件後，還另外提供了許多網路事件。

6.2 使用 WinME 開發應用程式指引

6.2.1 簡易換手策略程式

我們可以使用 WinME 來開發一個簡易的換手策略程式，假設目前我們的裝置有兩種界面，一種爲 802.11 WLAN 界面卡，一個爲 GPRS 界面卡。則我們的換手策略程式可以使用如下的設計：

假設 GPRS 是永遠連上線的。

1. 註冊 LINK_UP、LINK_DOWN 事件。

2. 當 WLAN 產生 LINK_DOWN 事件時，使用 WinME 提供的界面將預設閘道設爲 GPRS 界面卡。

3. 當 WLAN 界面卡產生 LINK_UP 事件時，使用 WinME 提供的界面去用 DHCP 幫 WLAN 界面卡取的 IP 位址和預設閘道。

6.2.2 架構在 SIP 上的 VoIP 程式

架構在 SIP 上的 VoIP 程式可能會想註冊 IP_CHANGE 和 DEFAULT_GATEWAY_CHNGE 的事件。當 IP_CHANGE 的事件產生時，VoIP 的應用程式就要準備要作 Re-Invite 的動作了，等到 DEFAULT_GATEWAY_CHANGE 產生後，就可以去作 Re-Invite 的動作了。

6.2.3 FTP 程式

FTP 應用程式可能會想註冊 LINK_UP、LINK_DOWN、IP_CHANGE、DEFAULT_GATEWAY_CHANGE 的事件。

當 LINK_DOWN 事件產生後，就停止傳輸的動作，等到 LINK_UP 事件發生時，可以嘗試續傳剛剛未傳完的資料，如果失敗的話，就等到 IP_CHANGE 的事件發生，當 IP_CHANGE 事件發生後，就要準備續傳了，之後等到 DEFAULT_GATEWAY_CHANGE 事件發生後，就可以開始續傳。

第 七 章 結論與未來工作

7.1 結論

現今使用者裝置已經可以配置多個無線網路模組，而在未來使用者將會愈來愈具有移動性，所以使用者所處的網路情況和以往固定在桌子上的電腦已經大不相同了。所以爲了提供使用者更好的使用經驗，應用程式需要針對目前網路的狀態來調整本身的行爲。

而本論文所提出的「支援具網路感知應用程式之中介軟體」可以讓應用程式的開發人員方便的去處理網路底層的相關工作，這對於撰寫換手策略相關程式是很有幫助的。

以往使用者空間的程式想要取得下層網路介面的狀態時，就不用不斷地使用系統呼叫來取得網路介面的狀態，並且和前一次取得的網路介面狀態作比對了，CPU 的時脈因此會浪費在這些多餘的系統呼叫上。

而本論文的中介軟體也提供了事件通知機制，因此網路事件產生時，應用程式可以直接被通知，而不用去輪詢，省下 CPU 的時脈。

最後我們在 Linux 系統下實作了我們的中介軟體並將之名命爲 WinME。

7.2 未來工作

本論文所提出的中介軟體，在未來可能可以增加更多的命令及事件，而 802.21 成爲標準後，也許也可以跟 802.21 整合在一起。

本論文僅提供了機制，未來希望能夠以此機制爲基礎，實作一套完整的行動管理員程式，幫助使用者在多網路介面的情況下作換手決策，對於其他的網路應用程式方面，使用該機制可以讓該應用程式對於網路狀態的改變更具有適應性。

參 考 文 獻

〔1〕 Dave Bakken, “Middleware”, Encyclopedia of Distributed Computing, Kluwer, to appear: http://www.eecs.wsu.edu/~bakken/middleware.pdf.

〔2〕 Jun-Zhao Sun, Jukka Riekki, Marko Jurmu, and Jaakko Sauvola, “Adaptive Connectivity Management Middleware for Heterogeneous Wireless Networks”, IEEE Wireless Communications, December 2005.

〔3〕 Sun J., Tenhunen J., and Sauvola J., “CME: a middleware architecture for network-aware adaptive applications”, In Proceedings 14th IEEE PIMRC2003, Beijing, China, 2003

〔4〕 Hawick K.A and James H.A., “Wireless Issues for a Mobility Management Middleware”, Submitted to CCN2002. Also DHPC Technical Report DHPC-111, August 2002.

〔5〕 Tian, Y. Frank, S. Tsaoussidis, V. Badr, H., “Middleware Design Issues for Application Management in Heterogeneous Networks”, Networks 2000. (ICON 2000).

〔6〕 Li B, Nahrstedt K., “A Control-Based Middleware Framework for Quality of Service Adaptations” , IEEE Journal of Selected Areas in Communication, 17(9): 1632～1650, 1999.

〔7〕 Anthony Sang-Bum, Jens Meggers, Gunnar Forsgren, Erno Kovacs, Michael Rosinus, “UMTS: A Middleware Architecture and Mobile API Approach”, IEEE Personal Communications, April, 1998.

〔8〕 Mika Ylianttila, "Vertical Handoff and Mobility - System Architecture and Transition Analysis", Faculty of Technology, Department of Electrical and Information Engineering, Infotech Oulu, University of Oulu, 2005.

〔9〕 Milind Buddhikot, Girish Chandranmenon, Seungjae Han, et al., "Design and Implementation of a WLAN/CDMA2000 Interworking Architecture", IEEE Communications Magazine, November 2003.

〔10〕 Hyosoon Park, et al., "Vertical Handoff Procedure and Algorithm between IEEE802.11 WLAN and CDMA Cellular Network", In Proceedings 7th CDMA International Conference, 217-221, Seoul, Korea, 2002.11.

〔11〕 Bianchi G., Tinnirello I., Scalia L., "Handover across Heterogeneous Wireless Systems: a Platform-Independent Control Logic Design", WPMC 2003.

〔12〕 Bernaschi M, Cacace F, Iannello G., "Vertical Handoff Performance in Heterogeneous Networks", In: ICPP' 2004 Workshops, Aug 2004.

〔13〕 Sebastien Pierrel, Topi Erlin, Janne Roslof, et al., “A Prototype for Policy Driven Control of Heterogeneous Network Access”, 2nd Workshop on Applications of Wireless Communications (WAWC'04), in conjunction with The 13th International Summer School on Telecommunications, Lappeenranta, Finland, August 4-6, 2004.
〔14〕 I. Tinnirello, L. Scalia, “Seamless Handover across Heterogeneous Wireless Networks: a Programmable Metric Approach”, SCI 2003, Luglio 2003, Orlando.
〔15〕 Helen J. Wang, Randy H. Katz, Jochen Giese, “Policy-Enabled Handoffs across Heterogeneous Wireless Networks”, Proc. of ACM WMCSA, 1999.
〔16〕 J. Rosenberg, H. Schulzrinne, G. Camarillo, et al., “SIP: Session Initiation Protocol”, IETF RFC3261, June 2002.
〔17〕 C. Perkins, Ed., “IP Mobility Support for IPv4”, IETF RFC3344, Nokia Research Center, August 2002.
〔18〕 Stevens W. R., “TCP/IP Illustrated, Volume I, The Protocols”, AWL, 1994.
〔19〕 George Xylomenos, George C. P, "Internet Protocol Performance over Networks with Wireless Links", IEEE Network 13(4) 55-63, 1999.
〔20〕 David. D. Clark, "The Structuring of Systems using Upcalls", ACM Symposium on Operating Systems, pp. 171-180, 1985.
〔21〕 Geoffrey Howard Cooper, “The Argument for Soft Layer of Protocols”, Tech. Rep. Tr-300, Massachussets Institute of Technology, Cambridge, May 1983.
〔22〕 Carneiro G., Ruela J., Ricardo M., “Cross-Layer design in 4G Wireless Terminals”, IEEE Wireless Communications, 11(2):7–13, April 2004.
〔23〕 Arjan Peddemors, Hans Zandbelt, Mortaza Bargh, “A Mechanism for Host Mobility Management supporting Application Awareness”, MobiSys’04, Boston, Massachusetts, USA, June 6-9, 2004.
〔24〕 Jinwei Cao, Dongsong Zhang, Kevin M. McNeill, “An overview of network-aware applications for mobile multimedia delivery”, IEEE International Conference on System Sciences, Hawaii, 2004.
〔25〕 Qi Qu, Yong Pei, XS Tian, “Network-Aware Source-Adaptive Video Coding for Wireless Applications”, IEEE Military Communicatians Conference, MILCOM 2004 – 2004.
〔26〕 Brian L. Tierney, Dan Gunter, Jason Lee, “Enabling Network-Aware Applications”, In 10th IEEE Symposium on High Performance Distributed Computing, 2001.
〔27〕 Nancy Miller, Peter Steenkiste, “Collecting Network Status Information for Network-Aware Applications”, IEEE INFOCOM 2000.

〔28〕 Jurg Bolliger, Thomas Gross, “A Framework-Based Approach to the Development of Network-Aware Applications”, IEEE transactions on Software Engineering, vol. 24 no. 5, pp 376-390, 1998.

〔29〕 J. Salim, H. Khosravi, A. Kleen, A. Kuznetsov, “Linux Netlink as an IP Services Protocol”, IETF RFC 3549, July 2003.

〔30〕 IEEE 802.21 WG, “Draft IEEE Standard for Local and Metropolitan Area Networks: Media Independent Handover Services”, July 2005.

〔31〕 T. Farnham, A. Gefflaut, A. Ibing, et al., “Toward Open and Unified Link-Layer API”, European Gollum Project, Proceedings of the IST Mobile and Wireless Summit 2005.

〔32〕 Ta-Juan Chou, “Design and Implementation of Driver-Level Network Event Notification Mechanism in Linux”, WIN Lab NCTU, 2005.

〔33〕 Shigeo Shioda, Takahiro Yagi, “A new approach to the bottleneck bandwidth measurement for an end-to-end network path”, Communications, 2005. ICC 2005. IEEE International Conference 2005.

〔34〕 Cao Le Thanh Man, Go Hasegawa and Masayuki Murata, “Available bandwidth measurement via TCP connection”, in Proceedings of IFIP/IEEE MMNS ’04 E2EMON Workshop, Oct. 2004.

〔35〕 Manish Jain, Constantinos Dovrolis, “End-to-end available bandwidth: measurement methodology, dynamics, and relation with TCP throughput”, ACM SIGCOMM Computer Communication Review Volume 32, Issue 4, October 2002.

〔36〕 Jurg Bolliger, Thomas Gross, Urs Hengartner, “Bandwidth Modeling for Network-Aware Applications”, in Proc. of IEEE INFOCOM, 1999.

〔37〕 A. Yegin, Ed., “Link-layer Event Notifications for Detecting Network Attachments”, IETF draft-ietf-dna-link-information-03, Samsung AIT, October 24, 2005.

〔38〕 Klaus Whhrle, Frank Pahlke, Hartmut Ritter, et al., “The LINUX Networking Architecture – Design and Implementation of Network Protocols in the Linux Kernel”, Chapter 5, 2002.

〔39〕 K. El Malki, Editor, “Low Latency Handoffs in Mobile IPv4”, IETF draft-ietf-mobileip-lowlatency-handoffs-v4-11, Athonet, 3 October 2005.

〔40〕 Hidetoshi Yokota, Akira Idoue, Toru Hasegawa, “Link Layer Assisted Mobile IP Fast Handoff Method over Wireless LAN Networks”, MOBICOM’02, Atlanta, Georgia, USA, September 23–28, 2002.

〔41〕 Niebert N., Prytz M., Schieder A., et al., “Ambient Networks: a Framework for Future Wireless Internetworking”, To appear Proc. IEEE 61st Semiannual Vehicular Technology Conference (VTC 2005 Spring), Stockholm, Sweden, May 30 - June 1, 2005.

〔42〕 Petri Mahonen and Tommi Saarinen, “Platform-Independent IP Transmission over Wireless Networks: The WINE Approach”, IEEE Personal Communications, December 2001.

〔43〕 Vijay T. Raisinghani, Sridhar Iyer, “Cross-Layer Feedback Architecture for Mobile Device Protocol Stacks”, IEEE Communications Magazine, January 2006.

〔44〕 Vijay T. Raisinghani, Sridhar Iyer, “ECLAIR: An Efficient Cross-Layer Architecture for Wireless Protocol Stacks”, World Wireless Cong., San Francisco, CA, May 2004.

〔45〕 Qi Wang, Abu-Rgheff, M.A., “Cross-layer signaling for next-generation wireless systems”, IEEE Wireless Communications and Network-ing Conference (WCNC 2003), Vol. 2 , pp. 1084-1089, 16-20 March 2003.

〔46〕 Qi. Wang, Abu-Rgheff M., ”A Multi-Layer Mobility Management Architecture using Cross-Layer Signaling Interactions”, In: Proceeding of EPMCC ´03 , Glasgow, Scotland, Apr 2003.

〔47〕 Sanjay. Shakkottai, Theodore S. Rappaort, Peter C. Karlsson, “Cross layer design for wireless networks”, IEEE Communications Magazine, October 2003.

〔48〕 Vijay T. Raisinghani, Sridhar Iyer., “User Managed Wireless Protocol Stacks”, ICDCS, Providence, RI, USA, May 2003.

〔49〕 Raisinghani, V.T.; Singh, A.K.; Iyer, S., “Improving TCP Performance over Mobile Wireless Environments using Cross Layer Feedback”, IEEE International Conference on Personal Wireless Communications, pp. 81–85, 15-17 Dec. 2002.

〔50〕 Byoung-Jo Kim, “A Network Service providing Wireless Channel Information for Adaptive Mobile Applications: Part I: Proposal”, ICC ’01, June 2001.

〔51〕 Brian Noble, “System Support for Mobile, Adaptive Applications”, IEEE Personal Communications, pp. 44-49, Oct. 2000.

〔52〕 Yulia lndrayani Wijata, Douglas Niehaus, “A Scalable Agent-Based Network Measurement Infrastructure”, IEEE Communicatin Magazine, September 2000.

〔53〕 Julie Harmer , “Mobile aware multimedia applications for UMTS: the ACTS On The Moveproject”, Personal, Indoor and Mobile Radio Communications, 1997.

〔54〕 Caripe, W., Cybenko, G., Moizumi K., “Network Awareness and Mobile Agent Systems, IEEE Communications Magazine”, 36(7): pp. 44-49, July, 1998.
〔55〕 Chen L.J., Sun T., Chen B., ”A smart decision model for vertical handoff”, In Proceedings 4th ANWIRE International Workshop on Wireless Internet and Reconfigurability (ANWIRE 2004), Athens, Greece, 2004.
〔56〕 Ian F. Akyildiz, Jlang X., Shantidev M. Georgia, “A Survey of Mobility Management in Next-Generation ALL-IP-BASED Wireless Systems”, IEEE Wireless Communications, August 2004.
〔57〕 Christos P., Kar A. Chew, Rahim T., “Multilayer Mobility Management for All-IP Networks: Pure Sip vs. hybrid SIP/Mobile IP”, In The 57th IEEE Semiannual Vehicular Technology Conference,, pages 2500 – 2504, New York, NY, USA, 22-25 April 2003.
〔58〕 Aust S., Proetel D., Fikouras N.A., et al., “Policy based Mobile IP handoff decision (POLIMAND) using Generic Link Layer Information”, In Proceedings 5th IEEE Int. Conf. Mobile and Wireless Communication Networks, Oct. 2003.
〔59〕 C. Prehofer, W. Kellerer, R. Hirschfeld, et al., “An Architecture Supporting Adaptation and Evolution in Fourth Generation Mobile Communication Systems”, JCN, Vol. 4 No. 4, December 2002.
〔60〕 Michael E. Kounavis, Andrew T. Campbell, “Design, Implementation and Evaluation of Programmable Handoff in Mobile Networks”, ACM Mobile Networking (MONET), Vol.6, pp. 443-461, Sept 2001.
〔61〕 Archan M., Subir D., Prathima A., “Application-Centric Analysis of IP-Based Mobility Management Techniques”, Journal of Wireless Communications and Mobile Computing, vol. 1, issue 3, Aug 2001.
〔62〕 IEEE 802.11 WG, Part 11, “IEEE Std 802.11-1999: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specification,” 1999.
〔63〕 R. Droms, “Dynamic Host Configuration Protocol”, IETF RFC 2131, Bucknell University, March 1997.
〔64〕 Jouni M., “HostAP Driver for Intersil Prism 2/2.5/3”, http://hostap.epitest.fi/
〔65〕 Jean T., "Wireless Extensions for Linux", http://www.hpl.hp.com/personal/Jean_Tourrilhes/Linux/Tools.html
〔66〕 ”NLANR/DAST : Iperf - The TCP/UDP Bandwidth Measurement Tool”, http://dast.nlanr.net/Projects/Iperf/
〔67〕 “戶空間於內核空間之間的通信接口”, http://www.kernelchina.org/linuxkernel/chapter3.pdf

附錄A. 中介軟體 WinME 說明文件

A.1. 命令

A.1.1. 系統命令

A.1.1.1. GET_ALL_NETWORK_INTERFACE

取得目前系統上的所有網路界面。

Table A-1 GET_ALL_NETWORK_INTERFACE 的 win_cmd 欄位說明

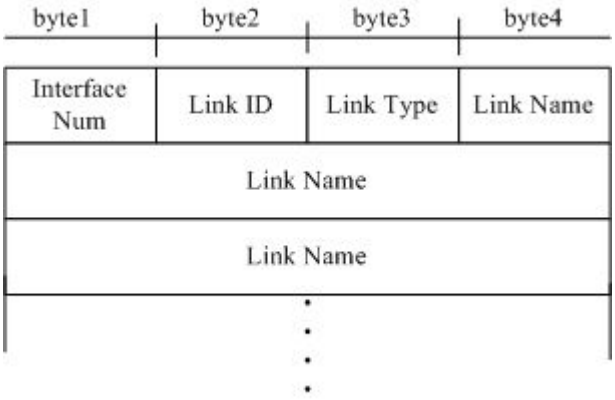
<table>
<tr><td>i8_cmd</td><td>GET_ALL_NETWORK_INTERFACE</td></tr>
<tr><td>pv_param</td><td>無作用</td></tr>
<tr><td>pv_value</td><td>傳回目前系統上的網路界面。
訊息格式如下:
 

Figure A-1 GET_ALL_NETWORK_INTERFACE 的 pv_value 格式
<table>
<tr><td>欄位名稱</td><td>長度</td><td>型別</td><td>欄位說明</td></tr>
<tr><td>Interface Num</td><td>1 byte</td><td>INT8</td><td>表示有幾個網路界面的資訊被傳回。</td></tr>
<tr><td>Link ID</td><td>1 byte</td><td>INT8</td><td>網路界面的 ID，用來分辨不同的界面。</td></tr>
<tr><td>Link Type</td><td>1 byte</td><td>INT8</td><td>網路界面的型態。</td></tr>
<tr><td>Link Name</td><td>9 bytes</td><td>CHAR[]</td><td>以 NULL 結尾的字串，爲系統內網路界面的名稱。在 Linux 系統可能爲 eth0、wlan0。</td></tr>
</table>
</td></tr>
</table>

A.1.1.2. GET_BATTERY_POWER

取得目前系統剩餘電力，適用於使用電池的裝置。

Table A-2 GET_BATTERY_POWER 的 win_cmd 欄位說明

i8_cmd	GET_BATTERY_POWER
pv_param	沒作用
pv_value	爲一 INT8 的值。介於 0 ~ 100。代表目前系統剩餘電力。

A.1.2. 硬體驅動程式命令

A.1.2.1. SET_POA

針對指定的網路界面卡設定接取點 (POA)。

Table A-3 SET_POA 的 win_cmd 欄位說明

<table>
<tr><td>i8_cmd</td><td>SET_POA</td></tr>
<tr><td>pv_param</td><td>指定網路界面要連接哪個接取點。
訊息格式如下:
 

Figure A-2 SET_POA 的 pv_param 格式
<table><tr><td>欄位名稱</td><td>長度</td><td>型別</td><td>說明</td></tr><tr><td>Link ID</td><td>1byte</td><td>INT8</td><td>指定網路界面卡。</td></tr><tr><td>POA Info</td><td></td><td></td><td>說定接取點資訊。接取點資訊會因爲網路界面卡的型態不同而有不同的值。</td></tr></table>以下列出不同網路界面卡型態時所要的接取點資訊:
LINK_TYPE_802_11:</td></tr>
</table>

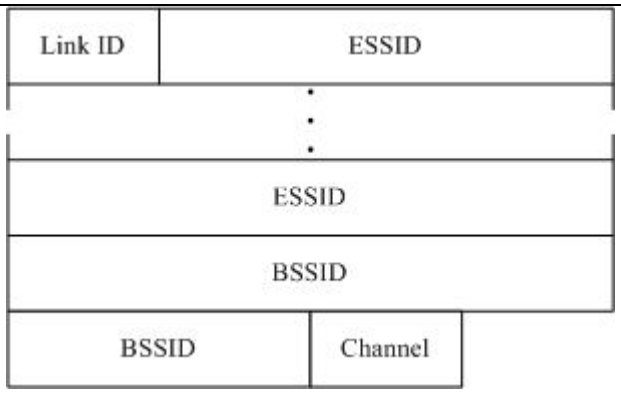


Figure A-3　802.11 網路下，SET_POA 的 pv_param 格式

Table A-4 802.11 網路下，SET_POA 的 pv_param 格式欄位說明

欄位名稱	長度	型別	說明
Link ID	1byte	INT8	指定網路界面卡。
ESSID	32bytes	CHAR[32]	表示 ESSID，爲 0 結尾的字串。如果沒填表示不改變目前設定。
BSSID	6bytes	CHAR[6]	表示 AP 的 BSSID。全填 0 表示不指定 BSSID。
Channel	1 byte	INT8	指定 AP 所在的 Channel。 0 表示不指定。

pv_value	無作用。

A.1.2.2. GET_POA

取得指定網路界面卡的接取點資訊。

Table A-5 GET_POA 的 win_cmd 欄位說明

i8_cmd	GET_POA
pv_param	爲一 INT8 的值。表示 Link ID。
pv_value	傳回指定網路卡的接取點資訊。在 802.11 中爲 AP 資訊。 訊息格式如下:  **Figure A-4 GET_POA 的 pv_value 格式** 欄位名稱 | 長度 | 型別 | 說明 Link Type | 1byte | INT8 | 表示網路界面卡的型態，接取點的資訊會因爲不同的型態而有不同的格式。

POA Info			接取點資訊。

以下列出不同 Link Type 時的接取點資訊:

LINK_TYPE_802_11:

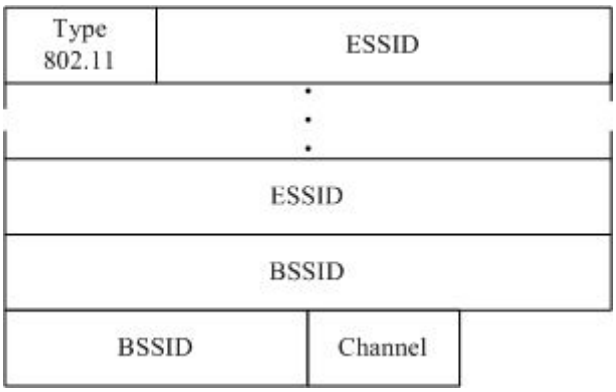


Figure A-5 802.11 網路下，GET_POA 的 pv_value 格式

Table A-6 802.11 網路下，GET_POA 的 pv_value 格式欄位說明

欄位名稱	長度	型別	說明
Link Type	1byte	INT8	爲 LINK_TYPE_802_11
ESSID	32bytes	CHAR[32]	表示 ESSID，爲 0 結尾的字串。
BSSID	6bytes	CHAR[6]	AP 的 BSSID。
Channel	1 byte	INT8	AP 所在的 Channel。

A.1.2.3. POA_SCAN

針對指定的網路界面下達掃描鄰近接取點 (POA) 的命令。POA (point of attachment) 在 802.11 的網路下爲AP (Access Point)。掃描結束後會由事件來通知，請參考 A.2.2.8。

Table A-7 POA_SCAN 的 win_cmd 欄位說明

i8_cmd	POA_SCAN
pv_param	爲一INT8 的值。表示Link ID，用來指定由哪個網路界面來作掃描的動作。Link ID的取得可參考 A.1.1.1。
pv_value	無作用。

A.1.2.4. GET_SIGNAL_STRENGTH

取得指定網路卡和所連接的 POA 的訊號強度。

Table A-8 GET_SIGNAL_STRENGTH 的 win_cmd 欄位

i8_cmd	GET_SIGNAL_STRENGTH
pv_param	爲 INT8 型別。 表示 Link ID，用來指定網路界面。
pv_value	爲 INT8 型別。 傳回訊號強度。以百分比爲單位。

A.1.2.5. SET_SIGNAL_THRESHOLD

設定訊號強度的門檻 (Threshold) 以百分比爲單位 100 爲最強，0 爲最弱 。當程式開發者不想要定期的去取得目前網路界面所連結的POA的訊號強度時，可藉由設定signal threshold來讓中介軟體在訊號強度跨過指定的門檻時以事件的方式通知。相關事件可參考 A.2.2.9。爲了防止訊號強度波動會一直跨過門檻，中介軟體提供設定AVOID_PINGPONG_VALUE的命令。

當 (目前訊號強度 > AVOID_PINGPONG_VALUE + SIGNAL_THRESHOLD) 時才會收到訊號強度大於門檻的事件。

當 (目前訊號強度 < SIGNAL_THRESHOLD – AVOID_PINGPONG_VALUE) 時才會收到訊號強度小於門檻的事件。

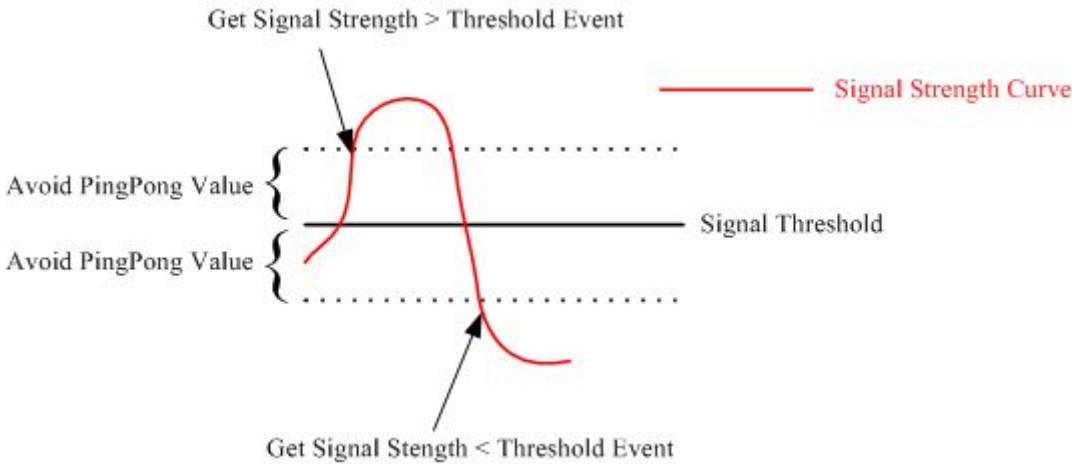


Figure A-6 訊號強度和訊號門檻等關係圖

Table A-9 SET_SIGNAL_THRESHOLD 的 win_cmd 欄位

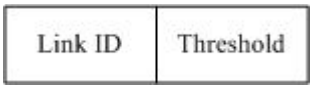
<table>
<tr><td>i8_cmd</td><td>SET_SIGNAL_THRESHOLD</td></tr>
<tr><td>pv_param</td><td>設定某張網路界面卡的訊號門檻。
訊息格式如下:
 

Figure A-7 SET_SIGNAL_THRESHOLD 的 pv_param 格式
<table><tr><td>欄位名稱</td><td>長度</td><td>型別</td><td>說明</td></tr><tr><td>Link ID</td><td>1byte</td><td>INT8</td><td>用來指定網路界面。</td></tr><tr><td>Threshold</td><td>1byte</td><td>INT8</td><td>設定訊號門檻。值爲 1 – 100。以百分比爲單位。</td></tr></table></td></tr>
<tr><td>pv_value</td><td>無作用。</td></tr>
</table>

A.1.2.6. GET_SIGNAL_THRESHOLD

查詢目前網路界面的訊號門檻。

Table A-10 GET_SIGNAL_THRESHOLD 的 win_cmd 欄位

i8_cmd	GET_SIGNAL_THRESHOLD
pv_param	爲 INT8 型別。 表示 Link ID，用來指定網路界面。
pv_value	爲 INT8 型別。 傳回訊號門檻。

A.1.2.7. SET_SIGNAL_AVOID_PINGPONG_VALUE

當程式設計者有設定訊號門檻 (A.1.2.5) 時，爲了防止訊號的波動一直產生訊號強度跨過訊號門檻的事件，可以設定防止乒乓值。相關運作方式請 A.1.2.5。如果沒有設定的話，預設值爲 0。

Table A-11 SET_SIGNAL_AVOID_PINGPONG_VALUE 的 win_cmd 欄

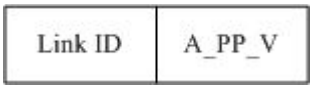
<table>
<tr><td>i8_cmd</td><td>SET_SIGNAL_AVOID_PINGPONG_VALUE</td></tr>
<tr><td>pv_param</td><td>設定某張網路界面卡的防止乒乓值。
訊息格式如下:
 

Figure A-8 SET_SIGNAL_AVOID_PINGPONG_VALUE 的 pv_param 格式</td></tr>
</table>

<table>
<tr><td></td><td><table>
<tr><td>欄位名稱</td><td>長度</td><td>型別</td><td>說明</td></tr>
<tr><td>Link ID</td><td>1byte</td><td>INT8</td><td>用於指定網路界面卡。</td></tr>
<tr><td>A_PP_V</td><td>1byte</td><td>INT8</td><td>指定 AVOID PINGPONG VALUE 值。值爲 0-100，以百分比爲單位。</td></tr>
</table></td></tr>
<tr><td>pv_value</td><td>無作用</td></tr>
</table>

A.1.2.8. GET_SIGNAL_AVOID_PINGPONG_VALUE

查詢 AVOID_PINGPONG_VALUE 值。

Table A-12 GET_SIGNAL_AVOID_PINGPONG_VALUE 的 win_cmd 欄位說明

i8_cmd	GET_SIGNAL_AVOID_PINGPONG_VALUE
pv_param	爲 INT8 型別。 表示 Link ID，用來指定網路界面。
pv_value	爲 INT8 型別。 傳回防止乒乓值。

A.1.2.9. SET_MTU

設定指定網路界面卡的最大傳輸單位 (Maximum Transmission Unit, MTU)。

Table A-13 爲一般常見網路界面的MTU：

Table A-13 一般常見網路界面的 MTU

網路界面	MTU (bytes)
EtherNet (一般的網路介面，這也是一般系統的預設值)	1500
PPPoE (ADSL 用的)	1492
Dial-up (modem)	576

Table A-14 SET_MTU 的 win_cmd 欄位

<table>
<tr><td>i8_cmd</td><td>SET_SIGNAL_AVOID_PINGPONG_VALUE</td></tr>
<tr><td>pv_param</td><td>設定指定網路界面卡的最大傳輸單位。
訊息格式如下:
 

Figure A-9 SET_MTU 的 pv_param 格式
<table>
<tr><td>欄位名稱</td><td>長度</td><td>型別</td><td>說明</td></tr>
<tr><td>Link ID</td><td>1byte</td><td>INT8</td><td>用於指定網路界面卡。</td></tr>
</table></td></tr>
</table>

	MTU	2bytes	UINT16	指定 MTU 值。
pv_value	無作用			

A.1.2.10. GET_MTU

取得指定網路界面卡上所設定的最大傳輸單位 (Maximum Transmission Unit, MTU)。

Table A-15 GET_MTU 的 win_cmd 欄位說明

i8_cmd	GET_MTU
pv_param	爲 INT8 型別。 表示 Link ID，用來指定網路界面。
pv_value	爲 UINT16 型別。 傳回最大傳輸單位 (MTU)。

A.1.2.11. GET_MAC_ADDRESS

取得指定網路界面卡的 MAC Address。

Table A-16 GET_MAC_ADDRESS 的 win_cmd 欄位

i8_cmd	GET_MAC_ADDRESS
pv_param	爲 INT8 型別。 表示 Link ID，用來指定網路界面。
pv_value	爲 CHAR[6] 陣列型別。 傳回 MAC Address。

A.1.2.12. GET_BANDWIDTH

取得指定網路界面卡目前所連線網路的頻寬。目前沒有實作該命令。

Table A-17 GET_BANDWIDTH 的 win_cmd 欄位

i8_cmd	GET_BANDWIDTH
pv_param	爲 INT8 型別。 表示 Link ID，用來指定網路界面。
pv_value	爲 UINT32 型別。 傳回中介軟體所測量的頻寬，單位爲 bytes。

A.1.2.13. GET_STATISTICS

取得指定網路界面卡的統計值。

Table A-18 GET_STATISTICS 的 win_cmd 欄位

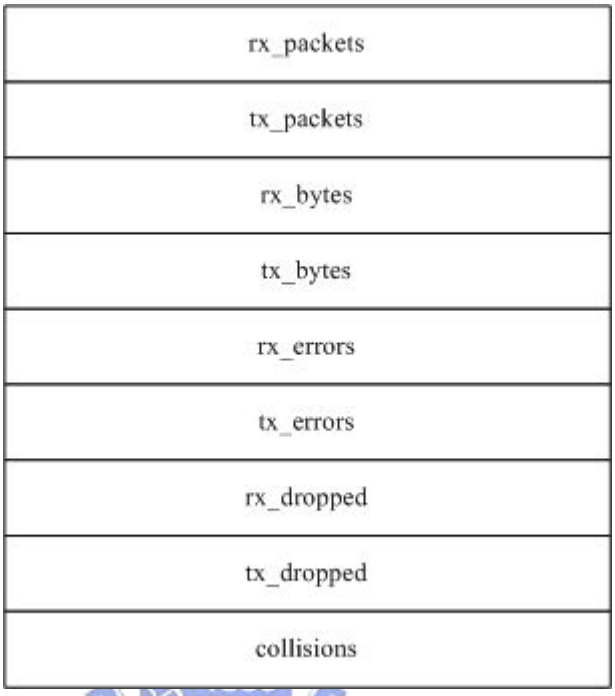
<table>
<tr><td>i8_cmd</td><td>GET_STATISTICS</td></tr>
<tr><td>pv_param</td><td>爲 INT8 型別。
表示 Link ID，用來指定網路界面。</td></tr>
<tr><td>pv_value</td><td>傳回統計資料。
訊息格式如下:
 

Figure A-10 GET_STATISTICS 的 pv_value 格式
<table>
<tr><th>名稱</th><th>長度</th><th>型別</th><th>說明</th></tr>
<tr><td>rx_packets</td><td>4 bytes</td><td>UINT32</td><td>收到的封包總數</td></tr>
<tr><td>tx_packets</td><td>4 bytes</td><td>UINT32</td><td>傳送的封包總數</td></tr>
<tr><td>rx_bytes</td><td>4 bytes</td><td>UINT32</td><td>收到的 bytes 總數</td></tr>
<tr><td>tx_bytes</td><td>4 bytes</td><td>UINT32</td><td>傳送的 bytes 總數</td></tr>
<tr><td>rx_errors</td><td>4 bytes</td><td>UINT32</td><td>收到的壞掉封包總數</td></tr>
<tr><td>tx_errors</td><td>4 bytes</td><td>UINT32</td><td>傳送封包失敗的次數</td></tr>
<tr><td>rx_dropped</td><td>4 bytes</td><td>UINT32</td><td>丟掉收到封包的次數，通常是系統內沒有記憶體可用了。</td></tr>
<tr><td>tx_dropped</td><td>4 bytes</td><td>UINT32</td><td>丟掉要傳送封包的次數，通常是系統內沒有記憶體可用了。</td></tr>
<tr><td>collisions</td><td>4 bytes</td><td>UINT32</td><td>產生碰撞的次數</td></tr>
</table>
</td></tr>
</table>

A.1.3. 網路層命令

A.1.3.1. SET_STATIC_IP

設定指定網路界面的網路位址 (IP ADDRESS)。

Table A-19 SET_STATIC_IP 的 win_cmd_ 欄位說明

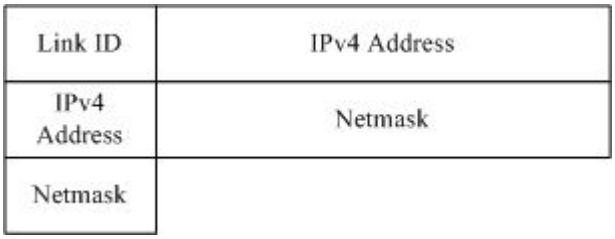
<table>
<tr><td>i8_cmd</td><td>SET_STATIC_IP</td></tr>
<tr><td>pv_param</td><td>設定指定網路界面卡的網路位址。
訊息格式如下:
 

Figure A-11 SET_STATIC_IP 的 pv_param 格式
<table>
<tr><td>名稱</td><td>長度</td><td>型別</td><td>說明</td></tr>
<tr><td>Link ID</td><td>1 byte</td><td>UINT8</td><td>指定網路界面卡。</td></tr>
<tr><td>IPv4 Address</td><td>4 bytes</td><td>UINT32</td><td>所要設定的網路位址。</td></tr>
<tr><td>Netmask</td><td>4 bytes</td><td>UINT32</td><td>設定子網路遮罩。 0 表示不更動原本設定。</td></tr>
</table>
</td></tr>
<tr><td>pv_value</td><td>無作用</td></tr>
</table>

A.1.3.2. SET_IP_BY_DHCP

使用 DHCP 爲指定的網路界面卡取得網路位址。

Table A-20 SET_IP_BY_DHCP 的 win_cmd 欄位

<table>
<tr><td>i8_cmd</td><td>SET_IP_BY_DHCP</td></tr>
<tr><td>pv_param</td><td>爲 INT8 型別。
表示 Link ID，用來指定網路界面。</td></tr>
<tr><td>pv_value</td><td>無作用</td></tr>
</table>

A.1.3.3. GET_IP

取得指定網路界面卡上目前所設定的網路位址。

Table A-21 GET_IP 的 win_cmd 欄位

i8_cmd	GET_IP
pv_param	爲 UINT8 型別。 表示 Link ID，用來指定網路界面。
pv_value	爲 UINT32 型別。 表示網路位址。

A.1.3.4. SET_NETMASK

設定指定網路界面卡的子網路遮罩。

Table A-22 SET_NETMASK 的 win_cmd 欄位

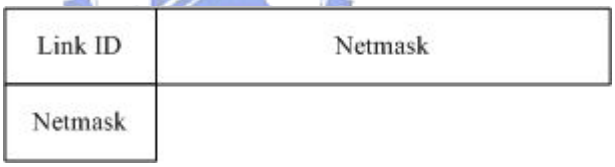
<table>
<tr><td>i8_cmd</td><td>SET_NETMASK</td></tr>
<tr><td>pv_param</td><td>設定指定網路界面卡的子網路遮罩。
訊息格式如下:
 

Figure A-12 SET_NETMASK 的 pv_param 格式
<table>
<tr><td>名稱</td><td>長度</td><td>型別</td><td>說明</td></tr>
<tr><td>Link ID</td><td>1 byte</td><td>UINT8</td><td>指定網路界面卡。</td></tr>
<tr><td>Netmask</td><td>4 bytes</td><td>UINT32</td><td>設定子網路遮罩。 0 表示不更動原本設定。</td></tr>
</table></td></tr>
<tr><td>pv_value</td><td>無作用</td></tr>
</table>

A.1.3.5. GET_NETMASK

取得指定網路界面卡的子網路遮罩。

Table A-23 GET_NETMASK 的 win_cmd 欄位

i8_cmd	GET_NETMASK
pv_param	爲 UINT8 型別。 表示 Link ID，用來指定網路界面。
pv_value	爲 UINT32 型別。 表示子網路遮罩。

A.1.3.6. SET_BROADCAST

設定廣播位址 (Broadcast) 的值，Broadcast 的算法會按照 IP 位址與 Netmask 的不同，會有不同的值。一般來說，未設定 broadcast，但 IP 位址與 Netmask 設定正確的話，網路還是可以正常動作，所以習慣上，常不會做 Broadcast 的設定。:

Table A-24 SET_BROADCAST 的 win_cmd 欄位

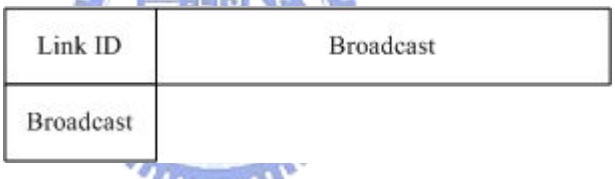
<table>
<tr><td>i8_cmd</td><td>SET_BROADCAST</td></tr>
<tr><td>pv_param</td><td>設定指定網路界面卡的廣播位址。
訊息格式如下:
 

Figure A-13 SET_BROADCAST 的 pv_param 格式
<table>
<tr><td>名稱</td><td>長度</td><td>型別</td><td>說明</td></tr>
<tr><td>Link ID</td><td>1 byte</td><td>UINT8</td><td>指定網路界面卡。</td></tr>
<tr><td>Broadcast</td><td>4 bytes</td><td>UINT32</td><td>設定廣播位址</td></tr>
</table></td></tr>
<tr><td>pv_value</td><td>無作用</td></tr>
</table>

A.1.3.7. GET_BROADCAST

取得指定網路界面卡的廣播位址。

Table A-25 GET_BROADCAST 的 win_cmd 欄位

i8_cmd	GET_BROADCAST
pv_param	爲 UINT8 型別。 表示 Link ID，用來指定網路界面。
pv_value	爲 UINT32 型別。 表示廣播位址。

A.1.3.8. SET_DEFAULT_GATEWAY

設定預設閘道，這也可以使用 SET_ROUTING_TABLE 的命令達成，不過預設閘道算是一個比較特別的路由表資料，所以獨立出一個命令。

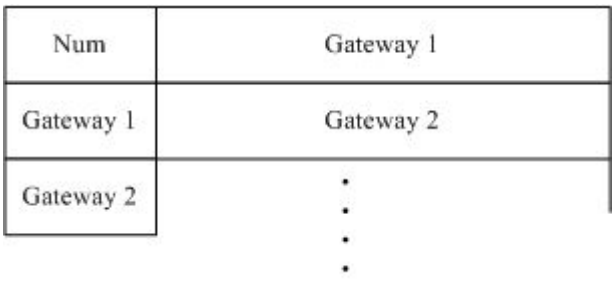
Table A-26 SET_DEFAULT_GATEWAY 的 win_cmd 欄位

i8_cmd	SET_DEFAULT_GATEWAY
pv_param	爲 UINT32 型別。 表示閘道位址，爲一 IPv4 位址。
pv_value	無作用

A.1.3.9. GET_DEFAULT_GATEWAY

取得目前系統上所設定的預設閘道。

Table A-27 GET_DEFAULT_GATEWAY 的 win_cmd 欄位

i8_cmd	GET_DEFAULT_GATEWAY
pv_param	無作用
pv_value	傳回目前系統上的預設閘道設定。 訊息格式如下:  **Figure A-14 GET_DEFAULT_GATEWAY 的 pv_value 格式** 名稱 | 長度 | 型別 | 說明 Num | 1 byte | UINT8 | 表示有多少 Gateway 資料。

	Gateway	4 bytes	UINT32	系統內設定的 Default Gateway 的 IPv4 位址。

A.1.3.10. SET_ ROUTING_TABLE

設定路由表資料。

Table A-28 SET_ROUTING_TABLE 的 win_cmd 欄位

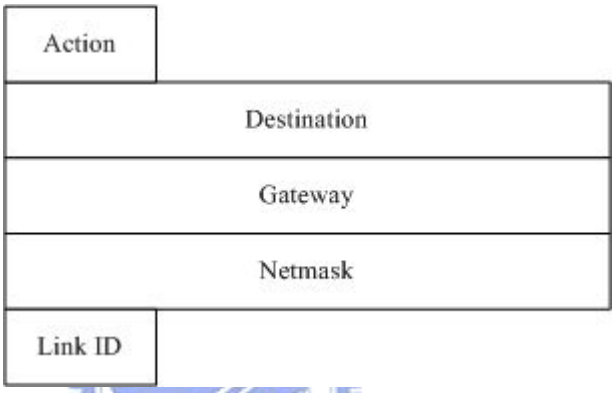
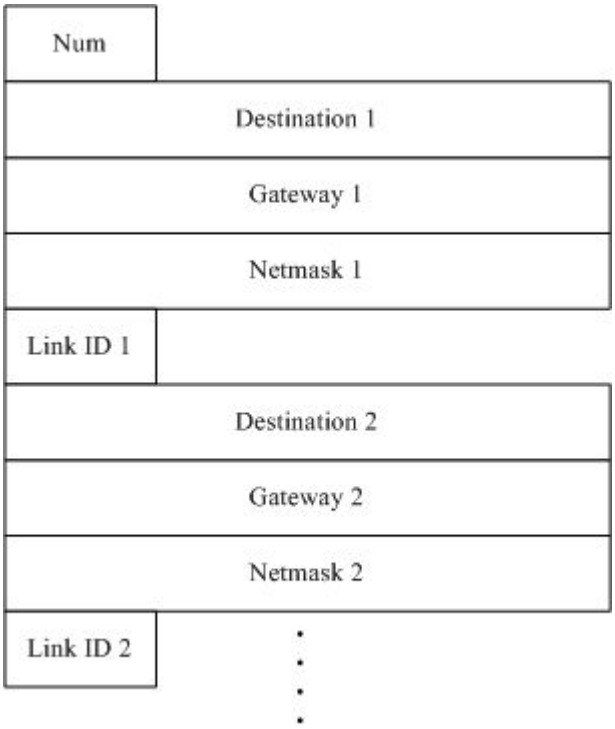
<table>
<tr><td>i8_cmd</td><td colspan="4">SET_ROUTING_TABLE</td></tr>
<tr><td>pv_param</td><td colspan="4">更改路由表資料。
訊息格式如下:
 

Figure A-15 SET_ROUTING_TABLE 的 win_cmd 欄位
<table><tr><td>名稱</td><td>長度</td><td>型別</td><td>說明</td></tr><tr><td>Action</td><td>1 byte</td><td>UINT8</td><td>表示處理的動作。
0: 刪除 (Delete)
1: 增加 (Add)</td></tr><tr><td>Destination</td><td>4 bytes</td><td>UINT32</td><td>目標，可以是 IPv4 位址也可以是網域.。
填 0 的話表示爲 Default。</td></tr><tr><td>Gateway</td><td>4 bytes</td><td>UINT32</td><td>該目標要經由哪一個閘道傳送。爲一 IPv4 位址。</td></tr><tr><td>Netmask</td><td>4 bytes</td><td>UINT32</td><td>該 Destination 的 Netmask。爲一 IPv4 位址。</td></tr><tr><td>Link ID</td><td>1 byte</td><td>UINT8</td><td>表示此設定所使用的網路界面卡。 0 表示不指定。</td></tr></table></td></tr>
<tr><td>pv_value</td><td colspan="4">無作用</td></tr>
</table>

A.1.3.11. GET_ROUTING_TABLE

取得路由表資料。

Table A-29 GET_ROUTING_TABLE 的 win_cmd 欄位

i8_cmd	GET_ROUTING_TABLE
pv_param	無作用
pv_value	傳回目前系統上的路由表設定。 訊息格式如下:  **Figure A-16 GET_ROUTING_TABLE 的 pv_value 格式** 名稱 / 長度 / 型別 / 說明 Num / 1 byte / UINT8 / 表示路由表有多少資料。 Destination / 4 bytes / UINT32 / 目標位址，可能爲 IPv4 位址或是網域.。 Gateway / 4 bytes / UINT32 / 該目標要經由哪一個閘道傳送。爲一 IPv4 位址。 Netmask / 4 bytes / UINT32 / 該 Destination 的 Netmask。爲一 IPv4 位址。 Link ID / 1 byte / UINT8 / 表示此設定所使用的網路界面卡。

A.1.4. 傳輸層命令

A.1.4.1. REGISTER_TCP_TX_STATE_EVENT

當 TCP 的傳送狀態改變時通知，這裡的狀態不是 Figure A-17 所顯示的TCP的內部實作狀態。而是TCP在避免擁塞時所處的狀態。

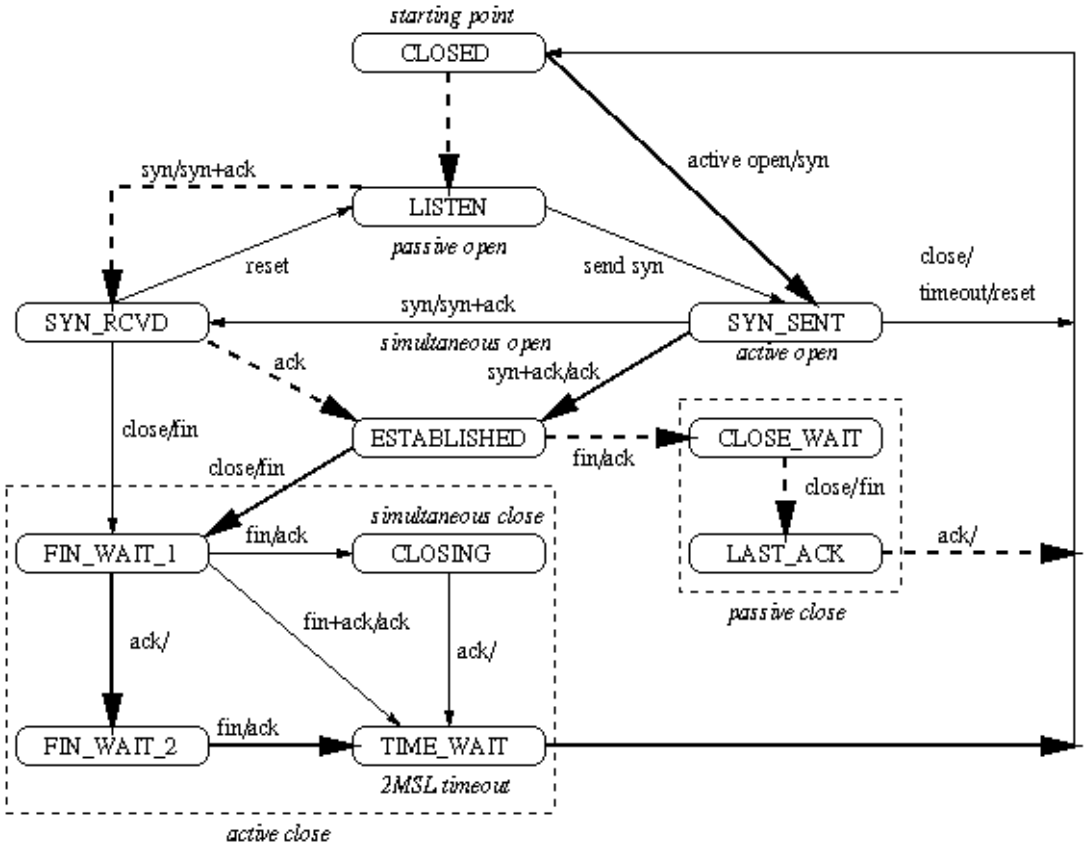


Figure A-17 TCP State Machine

Table A-30 REGISTER_TCP_STATE_EVENT 的 win_cmd 欄位

i8_cmd	REGISTER_TCP_TX_STATE_EVENT
pv_param	爲 INT32 型別。 表示 TCP 連線的 ID。在 Linux 下就是開 TCP 連線的 socket 傳回的值。
pv_value	無作用

A.1.4.2. GET_TCP_TX_STATE

取得 TCP 連線目前的傳送狀態。

Table A-31 GET_TCP_TX_STATE 的 win_cmd 欄位

i8_cmd	GET_TCP_TX_STATE
pv_param	爲 INT32 型別。 表示 TCP 連線的 ID。在 Linux 下就是開 TCP 連線 socket 傳回的值。
pv_value	爲 INT32 型別。 傳回目前的傳送狀態。

A.1.4.3. SET_TCP_TX_STATE

設定 TCP 連線的傳送狀態。

Table A-32 SET_TCP_TX_STATE 的 win_cmd 欄位

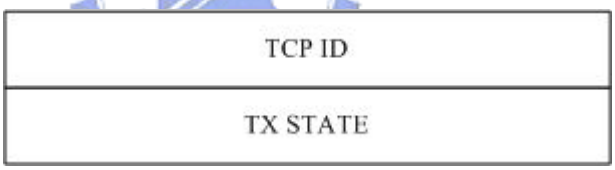
<table>
<tr><td>i8_cmd</td><td>SET_TCP_TX_STATE</td></tr>
<tr><td>pv_param</td><td>設定指定 TCP 連線的傳送狀態。
訊息格式如下:
 

Figure A-18 SET_TCP_TX_STATE 的 pv_param 格式
<table>
<tr><td>名稱</td><td>長度</td><td>型別</td><td>說明</td></tr>
<tr><td>TCP ID</td><td>4 byte</td><td>INT32</td><td>表示 TCP 連線的 ID。在 Linux 下就是開 TCP 連線 socket 傳回的值</td></tr>
<tr><td>TX STATE</td><td>4 bytes</td><td>INT32</td><td>設定傳送狀態。</td></tr>
</table></td></tr>
<tr><td>pv_value</td><td>無作用</td></tr>
</table>

A.1.4.4. GET_TCP_RETX_TIMEOUT

取得目前 TCP 連線多久沒收到 Ack 就重送封包。

Table A-33 GET_TCP_RETX_TIMEOUT 的 win_cmd 欄位

i8_cmd	GET_TCP_RETX_TIMEOUT
pv_param	爲 INT32 型別。 表示 TCP 連線的 ID。在 Linux 下就是開 TCP 連線 socket 傳回的值。
pv_value	爲 UINT32 型別。 傳回目前的重送封包 Timeout 值，單位爲 ms。

A.1.4.5. SET_TCP_RETX_TIMEOUT

設定目前 TCP 連線的重送封包的 Timeout 值。

Table A-34 SET_TCP_RETX_TIMEOUT 的 win_cmd 欄位

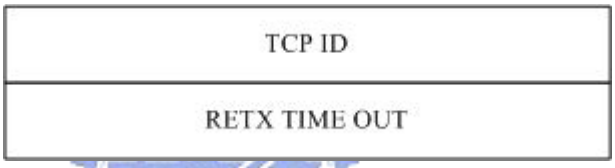
<table>
<tr><td>i8_cmd</td><td>SET_TCP_RETX_TIMEOUT</td></tr>
<tr><td>pv_param</td><td>設定指定 TCP 連線的重送封包的 Timeout 值。
訊息格式如下:
 

Figure A-19 SET_TCP_RETX_TIMEOUT 的 pv_param 格式
<table>
<tr><td>名稱</td><td>長度</td><td>型別</td><td>說明</td></tr>
<tr><td>TCP ID</td><td>4 byte</td><td>INT32</td><td>表示 TCP 連線的 ID。在 Linux 下就是開 TCP 連線 socket 傳回的值</td></tr>
<tr><td>RETX TIME OUT</td><td>4 bytes</td><td>UINT32</td><td>設定重送封包 Timeout 值，單位爲 ms。</td></tr>
</table>
</td></tr>
<tr><td>pv_value</td><td>無作用</td></tr>
</table>

A.1.4.6. GET_TCP_RTT

取得指定 TCP 連線在系統核心中的 Round Trip Time (RTT)。

Table A-35 GET_TCP_RTT 的 win_cmd 欄位

i8_cmd	GET_TCP_RTT
pv_param	爲 INT32 型別。 表示 TCP 連線的 ID。在 Linux 下就是開 TCP 連線 socket 傳回的值。
pv_value	爲 UINT32 型別。 傳回目前的 Round Trip Time (RTT)，單位爲 ms。

A.1.4.7. SET_TCP_RTT

設定目前 TCP 連線在系統核心中的 Round Trip Time (RTT) 值。

Table A-36 SET_TCP_RTT 的 win_cmd 欄位

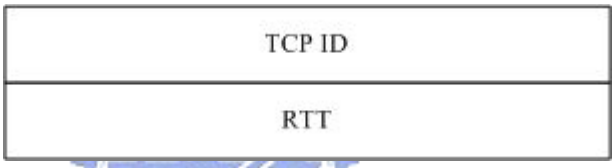
<table>
<tr><td>i8_cmd</td><td>SET_TCP_RTT</td></tr>
<tr><td>pv_param</td><td>設定指定 TCP 連線的 RTT 值。
訊息格式如下:
 

Figure A-20 SET_TCP_RETX_TIMEOUT 的 pv_param 格式
<table><tr><td>名稱</td><td>長度</td><td>型別</td><td>說明</td></tr><tr><td>TCP ID</td><td>4 byte</td><td>INT32</td><td>表示 TCP 連線的 ID。在 Linux 下就是開 TCP 連線 socket 傳回的值</td></tr></table></td></tr>
<tr><td>pv_value</td><td>無作用</td></tr>
</table>

A.1.4.8. GET_TCP_RECV_WIN

查詢目前 TCP 連線在核心內的 Receive Window 大小。

Table A-37 GET_TCP_RECV_WIN 的 win_cmd 欄位

i8_cmd	GET_TCP_RECV_WIN
pv_param	爲 INT32 型別。 表示 TCP 連線的 ID。在 Linux 下就是開 TCP 連線 socket 傳回的值。
pv_value	爲 UINT32 型別。 傳回目前的 Receive Window 大小，單位爲 byte。

A.1.4.9. SET_TCP_RECV_WIN

設定目前 TCP 連線在核心內的 Receive Window 大小。

Table A-38 SET_TCP_RECV_WIN 的 win_cmd 欄位

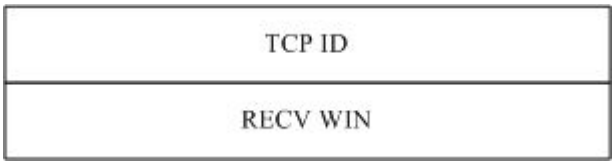
<table>
<tr><td>i8_cmd</td><td>SET_TCP_RECV_WIN</td></tr>
<tr><td>pv_param</td><td>設定指定 TCP 連線的 Receive Window 的大小。
訊息格式如下:
 

Figure A-21 SET_TCP_RECV_WIN 的 pv_param 格式
<table><tr><td>名稱</td><td>長度</td><td>型別</td><td>說明</td></tr><tr><td>TCP ID</td><td>4 byte</td><td>INT32</td><td>表示 TCP 連線的 ID。在 Linux 下就是開 TCP 連線 socket 傳回的值</td></tr><tr><td>RECV WIN</td><td>4 bytes</td><td>UINT32</td><td>設 Receive Window 值，單位爲 byte。</td></tr></table></td></tr>
<tr><td>pv_value</td><td>無作用</td></tr>
</table>

A.1.4.10. GET_TCP_CONG_WIN

查詢目前 TCP 連線在核心內的 Congestion Window 大小。

Table A-39 GET_TCP_CONG_WIN 的 win_cmd 欄位

i8_cmd	GET_TCP_CONG_WIN
pv_param	爲 INT32 型別。 表示 TCP 連線的 ID。在 Linux 下就是開 TCP 連線 socket 傳回的值。
pv_value	爲 UINT32 型別。 傳回目前的 Congestion Window 大小，單位爲 byte。

A.1.4.11. SET_TCP_CONG_WIN

設定目前 TCP 連線在核心內的 Congestion Window 大小。

Table A-40 SET_TCP_CONG_WIN 的 win_cmd 欄位

<table>
<tr><td>i8_cmd</td><td>SET_TCP_CONG_WIN</td></tr>
<tr><td>pv_param</td><td>設定指定 TCP 連線的 Receive Window 的大小。
訊息格式如下:
 

Figure A-22 SET_TCP_CONG_WIN 的 pv_param 格式
<table><tr><td>名稱</td><td>長度</td><td>型別</td><td>說明</td></tr><tr><td>TCP ID</td><td>4 byte</td><td>INT32</td><td>表示 TCP 連線的 ID。在 Linux 下就是開 TCP 連線 socket 傳回的值</td></tr><tr><td>CONG WIN</td><td>4 bytes</td><td>UINT32</td><td>設 Congestion Window 值，單位爲 byte。</td></tr></table></td></tr>
<tr><td>pv_value</td><td>無作用</td></tr>
</table>

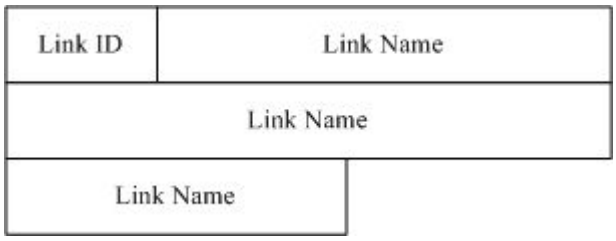
A.2. 事件

A.2.1. 系統事件

A.2.1.1. NETDEV_REGISTER

網路界面被新增到系統中。

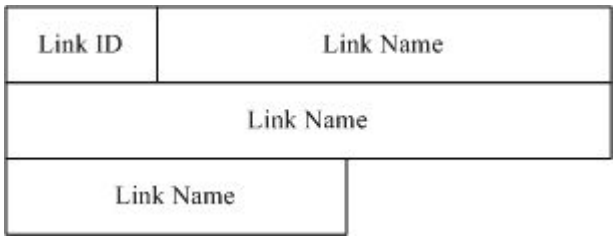
Table A-41 NETDEV_REGISTER 的 win_event 欄位

i8_event	NETDEV_REGISTER
pv_value	通知哪個網路界面被新增到系統中。 訊息格式如下:  **Figure A-23 NETDEV_REGISTER 的 pv_value 格式** 名稱 / 長度 / 型別 / 說明 Link ID / 1 byte / INT8 / 網路界面的 ID，用來分辨不同的界面。 Link Name / 9 bytes / CHAR[] / 以 NULL 結尾的字串，爲系統內網路界面的名稱。在 Linux 系統可能爲 eth0、wlan0。

A.2.1.2. NETDEV_UNREGISTER

網路界面卡從系統中移除。

Table A-42 NETDEV_UNREGISTER 的 win_event 欄位

i8_event	NETDEV_UNREGISTER
pv_value	通知哪個網路界面被從系統中移除。 訊息格式如下:  **Figure A-24 NETDEV_REGISTER 的 pv_value 格式** 名稱 / 長度 / 型別 / 說明 Link ID / 1 byte / INT8 / 網路界面的 ID，用來分辨不同的界面。 Link Name / 9 bytes / CHAR[] / 以 NULL 結尾的字串，爲系統內網路界面的名稱。在 Linux 系統可能爲 eth0、wlan0。

A.2.1.3. NETDEV_CHANGENAME

網路界面卡在系統內部的名稱改變。

Table A-43 NETDEV_CHANGENAME 的 win_event 欄位

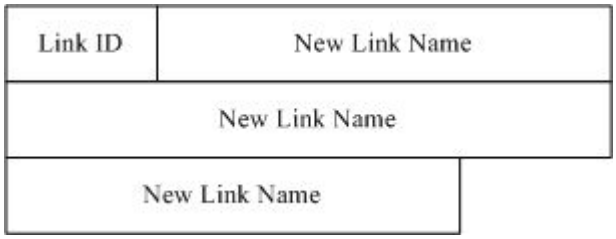
<table>
<tr><td>i8_event</td><td>NETDEV_CHANGENAME</td></tr>
<tr><td>pv_value</td><td>通知哪個網路界面的名稱改變了。
訊息格式如下:
 

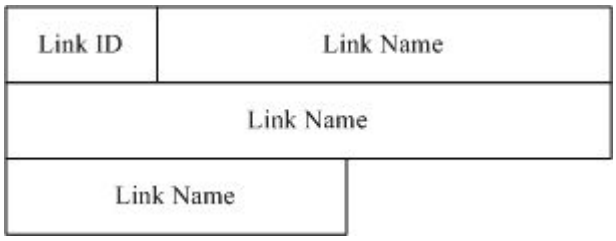
Figure A-25 NETDEV_CHANGENAME 的 pv_value 格式
<table>
<tr><th>名稱</th><th>長度</th><th>型別</th><th>說明</th></tr>
<tr><td>Link ID</td><td>1 byte</td><td>INT8</td><td>網路界面的 ID，用來分辨不同的界面。</td></tr>
<tr><td>New Link Name</td><td>9 bytes</td><td>CHAR[]</td><td>以 NULL 結尾的字串，爲改變過後的網路界面名稱。</td></tr>
</table></td></tr>
</table>

A.2.2. 硬體驅動程式事件

A.2.2.1. NETDEV_UP

某個網路界面被啓動了。

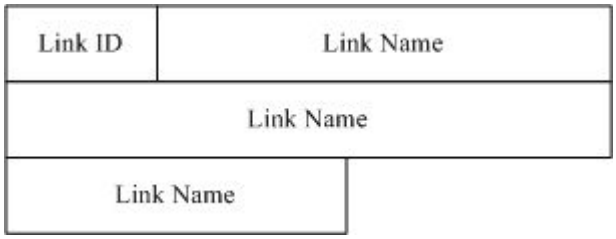
Table A-44 NETDEV_UP 的 win_event 欄位

i8_event	NETDEV_UP
pv_value	通知哪個網路界面被啓動了。 訊息格式如下:  **Figure A-26 NETDEV_REGISTER 的 pv_value 格式** 名稱 | 長度 | 型別 | 說明 Link ID | 1 byte | INT8 | 網路界面的 ID,用來分辨不同的界面。 Link Name | 9 bytes | CHAR[] | 以 NULL 結尾的字串,爲系統內網路界面的名稱。

A.2.2.2. NETDEV_DOWN

某個網路界面被關閉了。

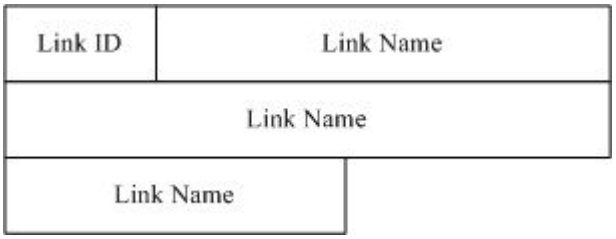
Table A-45 NETDEV_DOWN 的 win_event 欄位

i8_event	NETDEV_DOWN
pv_value	通知哪個網路界面被關閉了。 訊息格式如下:  **Figure A-27 NETDEV_DOWN 的 pv_value 格式** 名稱 | 長度 | 型別 | 說明 Link ID | 1 byte | INT8 | 網路界面的 ID,用來分辨不同的界面。 Link Name | 9 bytes | CHAR[] | 以 NULL 結尾的字串,爲系統內網路界面的名稱。

A.2.2.3. NETDEV_REBOOT

當網路介面偵測到硬體錯誤且需要重新啓動。

Table A-46 NETDEV_REBOOT 的 win_event 欄

i8_event	NETDEV_REBOOT
pv_value	通知哪個網路界面被偵測到硬體錯誤且需重新啓動。 訊息格式如下:  **Figure A-28 NETDEV_REGISTER 的 pv_value 格式** 名稱 / 長度 / 型別 / 說明 Link ID / 1 byte / INT8 / 網路界面的 ID，用來分辨不同的界面。 Link Name / 9 bytes / CHAR[] / 以 NULL 結尾的字串，爲系統內網路界面的名稱。

A.2.2.4. NETDEV_CHANGEMTU

網路介面更改 MTU (Maximum Transfer Unit)。

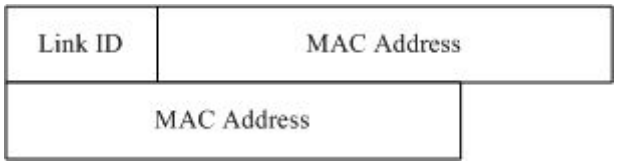
Table A-47 NETDEV_CHANGEMTU 的 win_event 欄位

i8_event	NETDEV_CHANGEMTU
pv_param	傳回哪個網路介面改更改的 MTU。 訊息格式如下:  Figure A-29 NETDEV_CHANGEMTU 的 pv_param 格式 欄位名稱 / 長度 / 型別 / 說明 Link ID / 1byte / INT8 / 表示網路界面卡。 MTU / 2bytes / UINT16 / 更改過後的 MTU 值。

A.2.2.5. NETDEV_CHANGEADDR

網路介面之硬體位址改變。

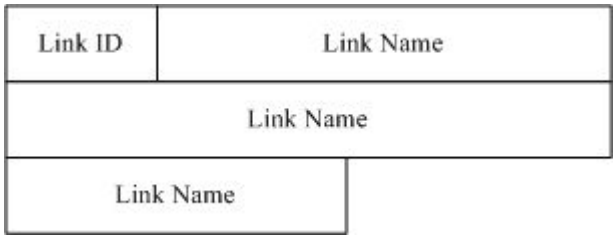
Table A-48 NETDEV_CHANGEADDR 的 win_event 欄位

i8_event	NETDEV_CHANGEADDR
pv_param	傳回哪個網路介面改更改的 MTU。 訊息格式如下:  **Figure A-30 NETDEV_CHANGEADDR 的 pv_value 格式** 欄位名稱 | 長度 | 型別 | 說明 Link ID | 1byte | INT8 | 表示網路界面卡。 MAC Address | 6bytes | char[] | 更改過後的 MAC Address。

A.2.2.6. LINK_UP

某張網路界面卡可以開始傳送資料封包了。通常爲網路界面卡和接取點 (POA) 有了連接。關於各種不同類型的網路界面卡在何時會產生 Link UP 的事件，可參考 IETF 的" Link-layer Event Notifications for Detecting Network Attachments"，目前還在 Draft 階段。這份文件有定義 ppp 界面、802.11 界面及 Etnernet 界面的 Link UP 情況。

Table A-49 LINK_UP 的 win_event 欄位

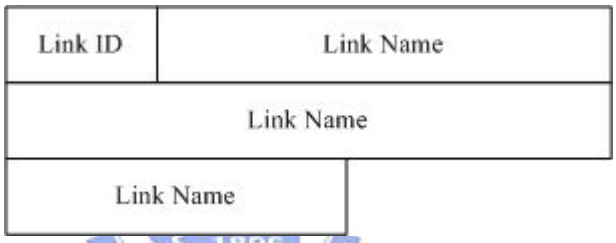
i8_event	LINK_UP
pv_value	通知哪個網路界面可以開始傳送資料封包。 訊息格式如下:  **Figure A-31 LINK_UP 的 pv_value 格式** 名稱 | 長度 | 型別 | 說明 Link ID | 1 byte | INT8 | 網路界面的 ID，用來分辨不同的界面。 Link Name | 9 bytes | CHAR[] | 以 NULL 結尾的字串，爲系統內網路界面的名稱。

A.2.2.7. LINK_DOWN

某張網路界面卡無法傳送資料封包了。通常爲網路界面卡和接取點(POA)的連接斷掉了。關於各種不同類型的網路界面卡在何時會產生 Link DOWN 的事件，可參考 IETF 的" Link-layer Event Notifications for Detecting Network Attachments"，目前還在 Draft 階段。這份文件有定義 ppp 界面、802.11 界面及 Etnernet 界面的 Link Down 情況。

Table A-50 LINK_DOWN 的 win_event 欄位

Table A-51 LINK_DOWN 的 win_event 欄位

i8_event	LINK_DOWN
pv_value	通知哪個網路界面沒有傳送資料封包的能力。 訊息格式如下:  **Figure A-32 LINK_DOWN 的 pv_value 格式**

名稱	長度	型別	說明
Link ID	1 byte	INT8	網路界面的 ID，用來分辨不同的界面。
Link Name	9 bytes	CHAR[]	以 NULL 結尾的字串，爲系統內網路界面的名稱。

A.2.2.8. POA_SCAN_COMPLETE

掃描接取點 (POA) 的動作完成了，並且傳回接取點的資訊。這個事件一定發生在 POA_SCAN的命令之後。

Table A-52 POA_SCAN_COMPLETE 的 win_event 欄位

i8_event	POA_SCAN_COMPLETE
pv_value	傳回接取點資訊。不同型態的網路界面有不同的資訊。由網路界面型態的欄位決定。 訊息格式如下:

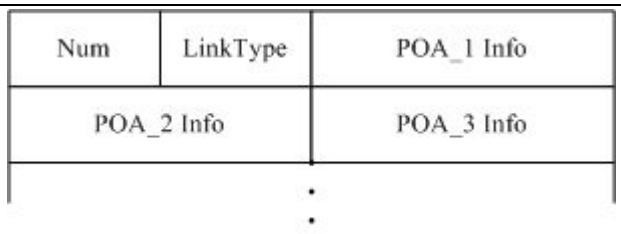


Figure A-33 POA_SCAN_COMPLETE 的 pv_value 格式

名稱	長度	型別	說明
Num	1 byte	INT8	表示有多少個接取點資訊被回傳。
LinkType	1 byte	INT8	網路界面的型態。
POA Info	不固定		由 LinkType 決定這邊的資訊。

以下列出不同 Link Type 時的接取點資訊:

LINK_TYPE_802_11:

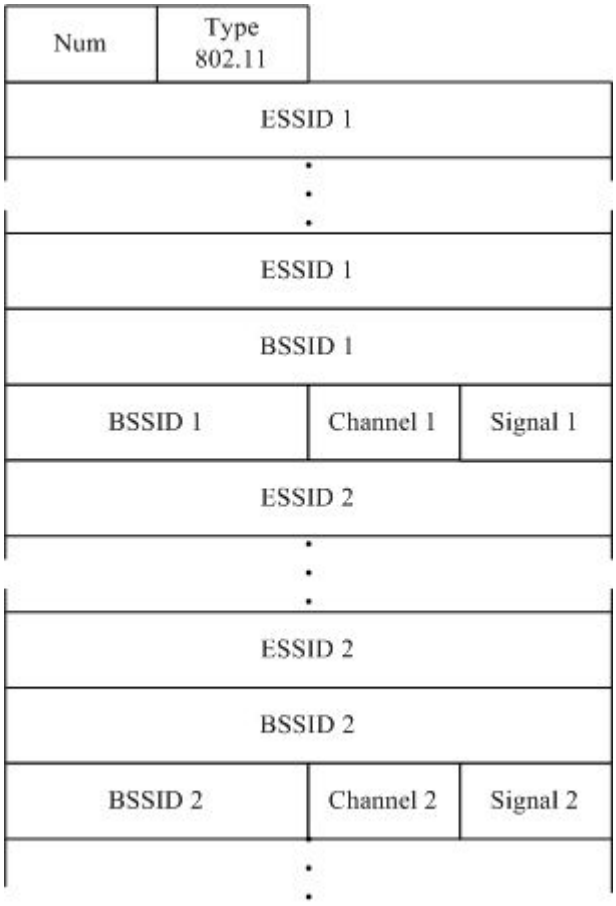


Figure A-34 802.11 網路下，POA_SCAN_COMPLETE 的 pv_value 格式

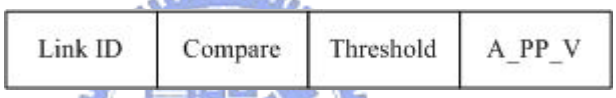
欄位名稱	長度	型別	說明
Num	1byte	INT8	表示有多少個接取點資訊被回傳。
Link Type	1byte	INT8	爲 LINK_TYPE_802_11

	ESSID	32bytes	CHAR[32]	表示 ESSID，爲 0 結尾的字串。
	BSSID	6bytes	CHAR[6]	AP 的 BSSID。
	Channel	1 byte	INT8	AP 所在的 Channel。
	Signal	1byte	INT8	傳回訊號強度。以百分比爲單位。

A.2.2.9. SIGNAL_STRENGTH_THRESHOLD

網路界面和所連接的POA訊號強度跨過程式設計者所指定的 THRESHOLD 時產生此事件通知程式設計者。可參考 A.1.2.5 及 A.1.2.7。

Table A-53 SIGNAL_STRENGTH_THRESHOLD 的 win_event 欄位

i8_event	SIGNAL_STRENGTH_THRESHOLD
pv_value	通知哪個網路界面卡所連接的接取點訊號強度跨過門檻了。 訊息格式如下:  **Figure A-35 SIGNAL_STRENGTH_THRESHOLD 的 pv_value 格式**

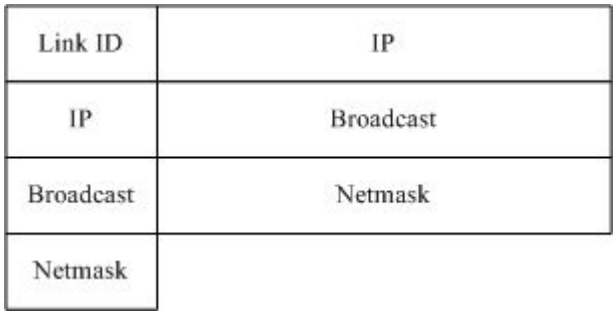
名稱	長度	型別	說明
Link ID	1 byte	INT8	網路界面的 ID，用來分辨不同的界面。
Compare	1 byte	INT8	表示訊號強度是大於或小於門檻。 -1: 小於 1: 大於
Threshold	1 byte	INT8	表示當初所設定的門檻。可參考 A.1.2.5。
A_PP_V	1 byte	INT8	當初所指定AVOID PINGPONG VALUE 值。可參考 A.1.2.7。

A.2.3. 網路層事件

A.2.3.1. IP_CHANGE

某個網路界面卡上的網路位址 (IP Address) 改變了。

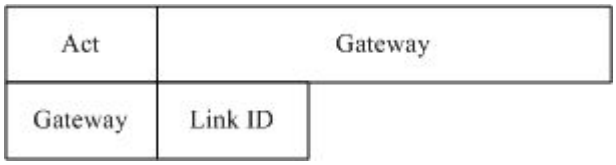
Table A-54 IP_CHANGE 的 win_event 欄位

i8_event	IP_CHANGE
pv_value	通知哪張網路界面卡的網路位址改變了。 訊息格式如下:  **Figure A-36 IP_CHANGE 的 pv_value 格式** 名稱 / 長度 / 型別 / 說明 Link ID / 1 byte / UINT8 / 表示網路界面卡。 IP / 4 bytes / UINT32 / 更改過後的網路位址。 NETMASK / 4 bytes / UINT32 / 子網路遮罩。 Broadcast / 4 bytes / UINT32 / 廣播位址。

A.2.3.2. DEFAULT_GATEWAY_CHANGE

預設閘道改變的事件，當程式設計者同時註冊這個事件和 ROUTING_TABLE_CHANGE 事件時，和預設閘道相關的，只會收到 DEFAULT_GATEWAY_CHANGE 事件。

Table A-55 DEFAULT_GATEWAY_CHANGE 的 win_event 欄位

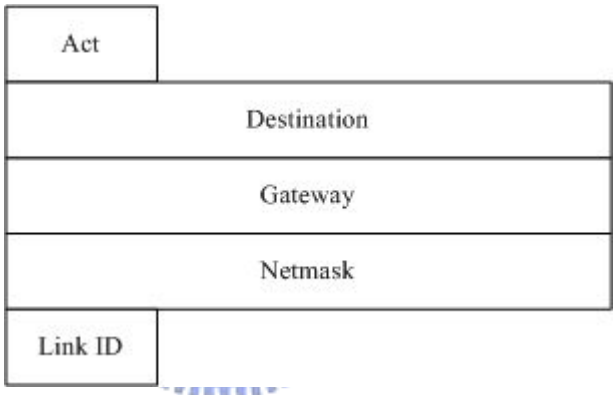
i8_event	DEFAULT_GATEWAY_CHANGE
pv_value	傳回預設閘道更改的資訊。 訊息格式如下:  **Figure A-37 DEFAULT_GATEWAY_CHANGE 的 pv_value 格式** 名稱 / 長度 / 型別 / 說明

	Act	1 byte	UINT8	表示預設閘道更動的動作。 0: 新增 (Add) 1: 刪除 (Del)
	Gateway	4 bytes	UINT32	Gateway 的 IPv4 位址。
	Link ID	1 byte	UINT8	經由哪個網路路界面卡。

A.2.3.3. ROUTING_TABLE_CHANGE

路由表資料改變的事件。

Table A-56 ROUTING_TABLE_CHANGE 的 win_event 欄位

i8_event	ROUTING_TABLE_CHANGE
pv_value	傳回路由表更動的資訊。 訊息格式如下:  **Figure A-38 ROUTING_TABLE_CHANGE 的 pv_value 格式**

名稱	長度	型別	說明
Act	1 byte	UINT8	表示處理的動作。 0: 刪除 (Delete) 1: 增加 (Add)
Destination	4 bytes	UINT32	目標。可能爲 IPv4 位址也或是網域.。
Gateway	4 bytes	UINT32	該目標要經由哪一個閘道傳送。爲一 IPv4 位址。
Netmask	4 bytes	UINT32	該 Destination 的 Netmask。爲一 IPv4 位址。
Link ID	1 byte	UINT8	經由哪個網路路界面卡。

A.2.4. 傳輸層事件

A.2.4.1. TCP_TX_STATE_CHANGE

TCP 傳送的狀態改變了。

Table A-57 TCP_TX_STATE_CHANGE 的 win_event 欄位

i8_event	TCP_TX_STATE_CHANGE
pv_value	爲 INT32 型別。 表示改變後的傳送狀態。