

追蹤與測試 Linux 路由器

楊佳欣 林盈達

投稿領域：Switch Routers

國立交通大學資訊科學系
新竹市大學路 1001 號

TEL：5712121 EXT 56667

E_MAIL：gis88567@cis.nctu.edu.tw, ydlin@cis.nctu.edu.tw

主要聯絡人：楊佳欣

摘要

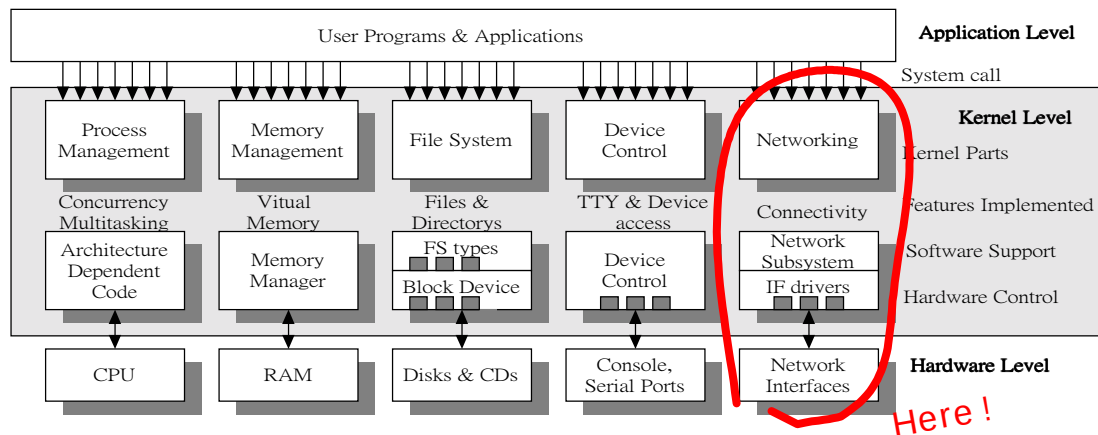
Linux 可以用來架設許多種類的伺服器，路由器為其中之一的選擇，而卻不用花到什麼錢，如果你有多餘的 PC 和網路卡、網路線等，甚至不用，而 Linux 本身更是免費的。那究竟這樣的做法有沒什麼缺點，適合在什麼環境之下，它的能力如何就是我們這篇文章的主題。我們花了些許時間實際架設 Linux 路由器並 trace 核心有關 IP 路由的原始程式以了解其運作，並試圖利用現有的函式量測這部份所花的時間，且利用 SmartBits 2000 測量 Linux 路由器的 latency 和 throughput。

關鍵字：Linux、router、routing、路由器、benchmark。

一、簡介

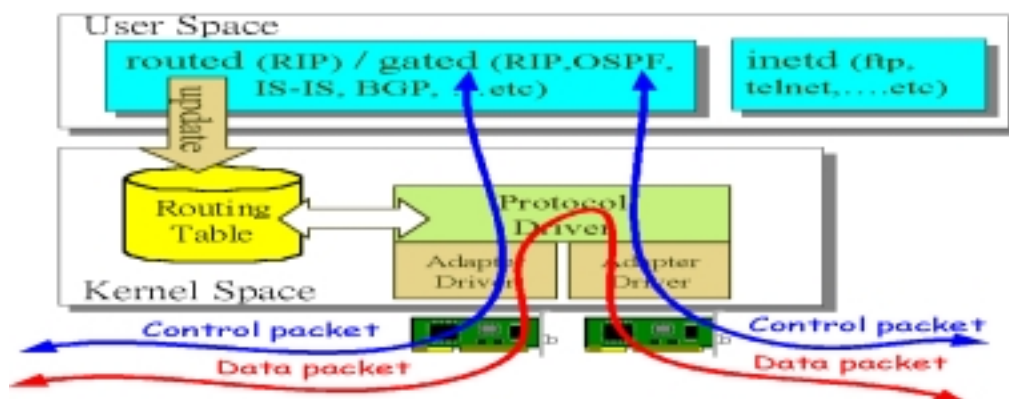
我們先介紹 Linux 的核心架構，及有關網路路由的模組[1][2]。圖一中最下層 Hardware Level 代表著網路卡，或者是 serial、parallel port。而在 Kernel Level 的 Hardware Control 有

各式介面的硬體驅動程式。再上層的 Software 提供著如 modem、token ring Ethernet 的基本傳輸協定。Kernel 中的 Networking 最主要在實做 TCP/IP 協定、當然也有關於 IPX、UUCP 等。



圖一、Linux 的模組

而從介面卡進入的封包到了 IP 層時便會被檢查是否是要給這台機器的。在一般的主機(不是路由器)上如果遇到不是要給自己的，則將其丟棄之，反之繼續送到上層處理。如果 OS 有做特別的設定，使之具有路由器的功能的話則根據路由表(routing table)選擇適當的路徑，這有時也被稱作轉送 (forward)。而路由表格大都數是以雜湊表 (hashing table) 的資料結構來組織，並利用適當的參數如 mask、destination address 來找到適當的位置。路由守護者 (routing daemon) 則負責實現各個路由協定 (routing protocols)、如 RIP 和 OSPF 等，且適當的根據溝通的結果來更改路由表的內容。此亦稱為動態路由 (dynamic routing)。其資料與控制封包之處理流程和各個模組之間的關係可以參考圖二。



圖二：Linux 路由流程及模組

在往後幾節我們會說明追蹤 Linux 原始碼的方法及一些心得以及我是如何測量各模組執行時間，並報告實際測試 Linux 路由器效能所得數據及結論。

二、追蹤和量測 Linux 的方法

在此我們檢視有關 Kernel 2.0.0 網路協定驅動程式的原始碼[3]，原始程式在核心程式樹（Kernel source tree）`/usr/src/linux/net/ipv4/`，例如在 `route.c` 中有許多重要的函式，如 `ip_rt_route`、`ip_rt_slow_route` 等，kernel 2.0.36 以前是如此，但 kernel 2.2.0 以後關於網路這部份的式碼已做了許多更動，不過對於查詢路由表的內容的 function 還是可以找到對應的如 `ip_route_output`、`ip_route_output_slow` 等，其它如 `ip_forward.c` 及 `ip_output.c` 等大概也都是如此。

核心安裝及更動

首先抓回自己要用的 kernel，在 <http://www.kernel.org/> 中有各版的 Kernel，台灣也有一個 mirror 站 <http://www.tw.kernel.org/>。抓回來後放到 `/usr/src/` 下再 `tar zxvf linux-2.2.X`，這樣就解開了。再做好 link，並設定核心參數 `cd /usr/src/linux`，`make menuconfig`、或其它如 `config`、`xconfig`。設定完進行編譯核心：`make dep`；`make clean`；`make zlilo`，請記得設定好 `/etc/lilo.conf` 這樣才不會白做工，如果 core image 太大也可以把 `make zlilo` 改成 `make bzlilo`，如果你設定時有將某個東西設定成為模組（module）的話還需要再執行 `make modules` 和 `make modules_install`。這些都可以在 `/usr/src/linux/Document/` 找到文件參考或者是在 Kernel HOWTO[4] 也有詳細的介紹。以後如果我們修改 Kernel 原始碼某個檔案時只需要 `make zlilo` 既可，而不是什麼事都再重新來一次，那要花很多時間的，請記住。

查詢文件、指令及檔案[6]

接下來你最好對 Linux 有些概念，如指令、檔案系統等。在 `make menuconfig` 時有個選項 `Extra Documents` 如果你選了的話則在 `/usr/doc/` 下會有許有的 HOWTO 或 LDP

的文件可供你查詢。一般如果我們對某個程式或函式不太了解時可以使用如 `#man [指令]` 或 `#man [函式]` 來查詢相關的資料，同時我們也可使用 `#man -k [關鍵字]` 來查詢相關的指令或函式（#代表提示指令輸入的符號）。現在除了 `man` 之外我們也可以使用 `info` 來查詢，使用方法跟 `man` 差不多。

那要如何查詢在檔案中的關鍵字、如變數或函數名稱呢？我們可以使用 `grep` 指令。用法如 `#grep [關鍵字] [檔案名]` 或使用萬用字元查詢在某個目錄下的所有檔案中有沒有任一檔案中有符合的，用法如 `#grep [關鍵字] *`，例如我們想在 `/usr/src/linux/net/ipv4` 下找尋有使用 `ip_rt_route` 函式的地方，那我們只要在那個目錄下執行 `#grep ip_rt_route *.c` 就可以了。那如果我們要知道某個函式或變數是在那被定義及實作的，這通常不是一件簡單的工作，因為在核心程式碼樹中的什麼地方都有可能，`.c` 檔中也有可能是在 `.h` 檔中，在那個目錄中可能也是個問題，這可能就要多花點時間。我們可以用下面這行來完成 `#find . | xargs grep ip_rt_route`，這行指令的功用在於列出目前工作目錄（.）下的檔案名稱再經由管線（|）送到 `xargs` 使之成為 `grep` 的參數，據此我們可以得知現行工作目錄有無任意檔案可以找到符合的字串。如果你想要更了解 `grep`、`find` 及 `xargs` 等最好是用 `man` 指令查詢手冊。同時你也可以使用 `locate` 指令查詢任意檔案在什麼位置。例如 `#locate route.h` 這樣你就可以查到從根目錄以下檔名為此的有那些及其路徑，不過前提是你是以系統管理者（root）的身分執行。

程式除錯工具

printk—Kernel 中要利用 `printk` 來將訊息列印出來，與 `printf` 不同的是，他沒有處理浮點數的能力，另外就是每一個 `printk` 還伴隨著一個 `log level`，可以說就是此 `message` 的等級。Linux 系統有一 `klogd` daemon，可以紀錄 kernel 的 `message`。如果系統上尚有 `syslogd`，就可以很方便地攔截 `klogd` 的 `message`，並根據不同的 `log level` 紀錄在不同的檔案或 show 在不同的 console。`syslogd` 的設定檔是 `/etc/syslogd.conf`，我們將其中 `kern.*` 的那行加上註解，並加上一行

```
"kern.=info          /var/log/kern_info"
```

就可以將所有 log level 為 KERN_INFO 的 message 寫在/var/log/ker_info 的檔案。使用 printk 的方法很簡單，其格式為 printk(loglevel “format string”)。在核心實作 TCP/IP 的程式碼中加入適量的 printk 函式呼叫，即可得到一些結果。

kdebug – 在 trace 程式時我們通常會用 debugger 來設 breakpoint，並 step by step 地去執行程式，然而在 debug 對像是 kernel 時，可就享受不到這種便利了。畢竟 debugger 也是 user space 的 application，要 debug 到 kernel space，就必須要透過一些其他的手段。kdebug 是一個好用的工具，其利用 gdb 的 remote debugging interface 在 run-time 來做到一些 debug 的動作，像是更改欲偵測對象之 data、呼叫函式（如呼叫 printk 印出某個變數目前的值）等。雖然 kdebug 可以在 run-time 做到上述的動作，卻仍不行做到設定中斷點或是單步執行，因為一台機器總不能邊把自己的 kernel 停掉，卻要他繼續執行 debugger 來看一些變數值等類的事。要做到這樣的事可以透過 remote debug 的方式，一台電腦跑 gdb，一台電腦跑 kernel，兩台用 RS-232 串連起來溝通。這種方法較複雜，詳情請參考[5]。

網路設定工具[7]

雖然在大多數情形下在安裝 Linux 時，安裝程式便會自動偵測你的網路卡並詢問相關網路設定，不過如果像我們這樣需要做一些比較複雜的設定或者想要了解 Linux 是如何運作的那就需要對這些工具非常了解，因為你常常會需要它們的幫助，不管你是自願還是被迫的。

ifconfig–可以用來設你的網路介面卡。

route–設定增加或減少路由。

netconf–選單式的介面可以設定大多數網路相關設定。

網路偵錯工具

ping–可以讓你了解某台機器是不是活在網路上，或是網路是不是出了什麼問題。

traceroute–讓你了解到達另一個主機時中間經過了那些路由轉送，花了多少時間。

netstat–可以印出路表，另外也可以讓你了解各種封包進出的統計資料。

tcpdump-可以印出經過本機上某個介面的封包標頭。

Sniffit[8]-LAN 上的網路分析工具。

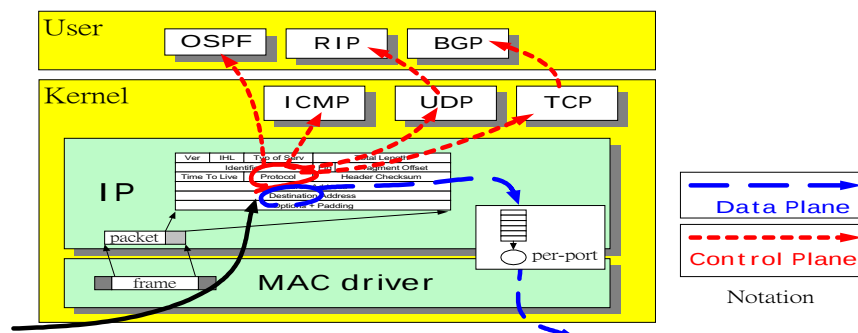
以上所提到的指令或工具大都都有手冊或說明檔可以參考。但其實在 Linux 上還有不少工具可以使用，不用花錢而且程式碼公開，如果你找不到適合自己用的工具可以到網路上問問或者是自己發展。

三、封包處理流程

在簡介時我們粗略地提到網路的模組，現在我們要更進一步說明這些封包是如何被這協定模組所處理。圖三為一個路由器中各個模組的簡圖，其中長虛線代表的一般的資料封包流程而短虛線則代表要給此本機的控制封包。一個封包我們接收進來後，便查看一下封包標頭的目的地地址欄位，如果目的地地址並不是自己的話，那我們需要查詢一下路由表決定一下這個封包應該要轉送到那介面的佇列中才是，至於在佇列中如何排班

(scheduling) 這可能會和 tos 欄位或一些排班法則有關。在設定 Kernel 時，如果你選擇了有關 QoS 的選項，則有為數不少的排班法則可供選則。你可以在文件[9]中找詳細的說明。

如果封包是要給自己的，那麼路由器就會在更進一步的分析是不是屬於 ICMP、UDP 或 TCP。如果是 UDP 的話，就看看是不是 RIP 的封包，是的話就交由路由守護者處理了，這可在 /etc/service 中得到。同樣地，BGP 協定則是屬於 TCP，OSPF 則是直接就在 IP 上。ICMP 更不用說就直接在 Kernel 中處理了。



圖三、封包處理示意圖

圖 1 為封包處理的流程圖。收到一個封包後便檢查目的地地址，如果給自己的，在大多數路由器中應該都跟實現路由協定或路由控制有關，所以我們就把它們往上層送。其它的封包應該就只是一般需要我們轉送的封包，先檢查 TTL，如果變成 0 的話丟棄封包並送出 ICMP time exceeded 的訊息給 IP 的來源地址，其它就是一般路由的過程了。在下面一節，我們會詳細介紹在 Linux 下的路由，包路由表的結構以及和其它模組之是如何溝通的。

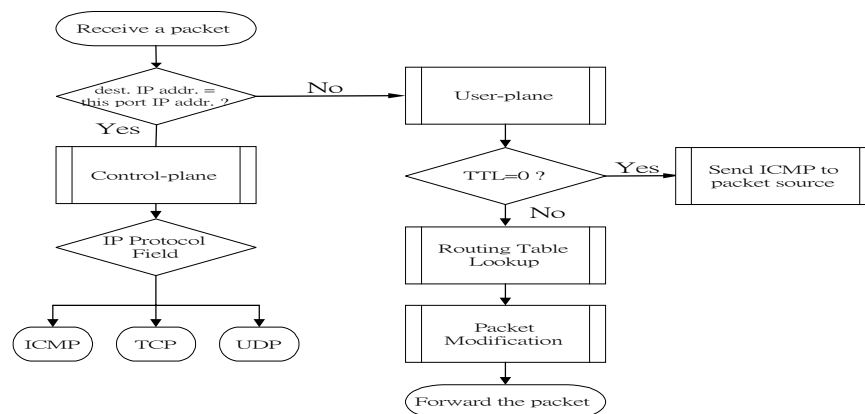


圖 1、封包處理流程

四、Linux 下的 IP 路由

在 IP 要選擇路由時，便會想要知道這個目的地地址是否可以到達？如果是可到達的話，要哪路比較好？如果有很多路可以的話，那會比較好。而這些問題的解答主要都要查詢路表來完成的。因為每個封包在傳送出去的過程會查詢，所以大大小小可以減少所花的時間。以下我們追蹤 Linux Kernel 2.0.X 下 IP 路由是如何實作的，我們先討論路由表格是如何由路由守護者決定及其使用的協定方法，再檢視路由表格查詢的過程。

在 Linux 下的路由守護者

當安裝 Linux 時，不少人看到這件時都到些許，更有不少人做為一台路由器就必須安裝個路由守護者，這樣路由器才能發功效。其實路由可分為態路

由（static routing）和動態路由（dynamic），動態路由簡單地在開機時由系統設定檔，像是 `/etc/rc.d` 下的 rc script，設定介面、路由或者是系統管理者根據一些資訊做適當的設定。資訊可能是根據某些方法線計得到的。而動態路由才需要守護者的加入用以實現一些路由方法。下面我們就為各位介紹在 Linux 中可執行的路由守護程式。routed[10]最主要用以實現 RIP 協定。一開始是由加州 Berkeley 分校所設計，伴隨著 BSD 的流行而被許多人使用。而 gated[11]支援多種路由協定。目前版本主要分為 unicast、multicast、ipv6 等，不過只有 version 3 提供原始碼，其它需要有會資格才可以下載。gated 同時也支援一些網路管理協定和 MIB。設定上有點複雜，如果有必要再試試看。Zebra[12]是一個新的 GNU General Public License 的路由協定實現軟體。特點在於其模組化的結構。每個協定都用個守護者來處理。下表比較各守護者所提供的路由協定。雖然 gated 只有版本 3 才提供原始碼，不過我們還是將其它版本加入比較，供大參考用。

	Hello	RIP	OSPF	IS-IS	SLSP	EGP	BGP	DVMRP	IGMP	PIM-SM	PIM-DM	GUM	CBT	IDRP
routed		V1												
gated, V3	Y	V1,2	Y	Y	Y	Y	V3,4							
gated, V4(Unicast)	Y	V1,2	Y	Y	Y	Y	V3,4	Y						
gated, V5(Multicast)		V1,2				Y		Y	Y	Y	Y	Y	Y	
gated, V6(IPv6)		ng	V6				V4++			v6	v6			v6
Zebra		Y	Y				V4							

表一、Linux 下各路由守護者所支援的協定

FIB 或是路由快取（routing cache）？

在 Kernel 中主要是由 `ip_rt_slow_route` 這個反解函式來解析封包應是如何路由，它會查詢 FIB（Forwarding Information dataBase）的資料項。而 FIB 這個 database 會將所有的路由資訊精心放好，不過為能加速查詢另外還有個路由快取（routing cache）的結構，和大多數的快取的實做方式一樣只會經常用到的資料，如果一項資料一段時間沒用到的話就會被刪除（expire），而本身也會用 LRU 方法重新排列資料項，這個查詢的過程如圖 1 所示。

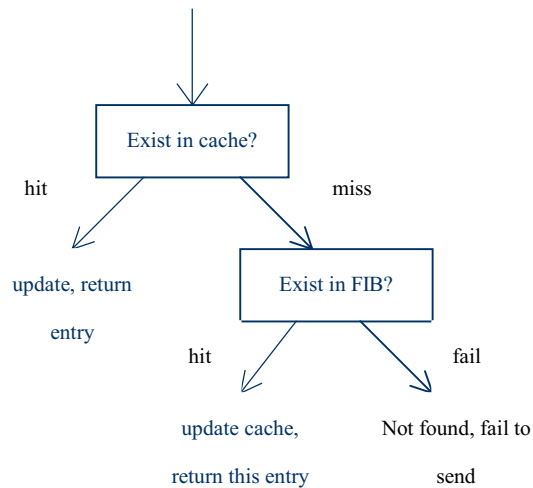


圖 1：路由查詢流程圖

路由快取的結構

圖 1 的流程由 `ip_rt_route` 實作，而 `ip_rt_route` 什麼時候會被呼叫呢？例如 `ip_output.c` 中的 `ip_build_header` 或者是 `ip_forward.c` 中的 `ip_forward` 等需要知道路由資訊時。查詢一開始時會先查詢路由表中的資料。用目的地址的後面兩個 byte 雜湊到 `ip_rt_hash_table` 中的某個串列，使用後兩個 byte 的原因是在大多數情形下不同封包的這兩個 byte 的值都不相同，因而可以分到雜湊表中不同的位置。經雜湊得到某個串列的位置再次比較串列中的資料項有沒有符合的，如果有的話便修改一些欄位，如參考次數、上次使用的時間、使用情形等，如果在路由表中找不到的話此時便要到 FIB 中查詢了，FIB 中存放所有的路由資訊，如果找到符合的資料項的話，除了傳回這筆資料外，並且將其加上對應的資料項到路由表中。最後如果在路由表和 FIB 中都找不到符合的資料項的話就回傳 NULL，這樣呼叫的人就知道沒有路由了。路由表的結構為一般雜湊表，如圖 2。

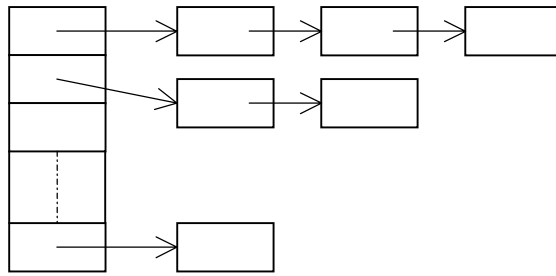


圖 1：路由表結構

FIB 的結構

而 FIB 的結構就比路由表複雜多了，每個 subnet 都由一個 fib_zone 資料結構對應，而這個資料結構都被 fib_zones 這個雜湊表所指到，雜湊表使用 subnet mask 做為參數進行雜湊。所有相同的子網域的路由表由一組 fib_node 和 fib_info 的資料結構所組成，當資料項數不是很多時由 fib_zone 中的 fz_list 指向這些路由資料，當超過一定數目時就雜湊這些個路由資訊並由 fib_zone 中的 fz_hash_table 所指到，也就是一個太長的串列經由雜湊分成好幾個較短的串列，使得尋找的過程更為快速。fib_node 主要是有關 destination、metric。而 fib_info 關心的是 device 和 gateway。圖 2 簡單地呈現 FIB 的資料結構。FIB 的表是一個非常複雜的資料結構，也許它對資料的存儲方式的設計很巧妙，但並不適合在每次要傳送時查詢路資訊時用，這也是我們為什麼需要路由表的原因。

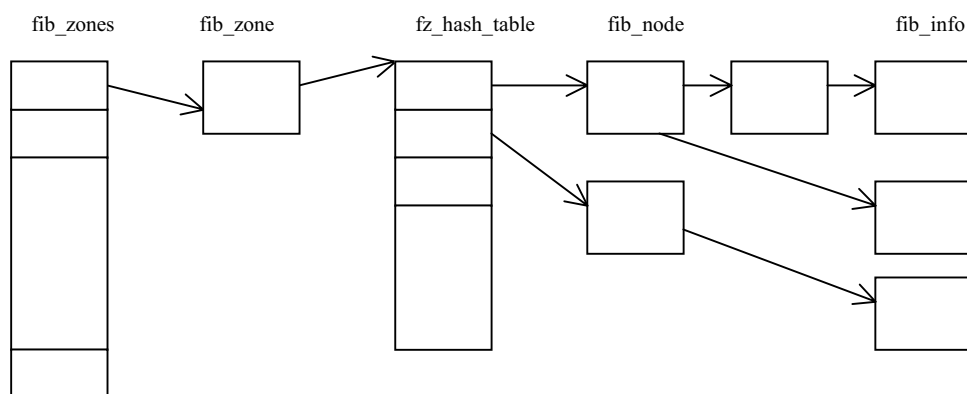


圖 2：FIB 的結構

如何讀取及寫入路由呢？

然而我們要如何才能對路由表進行讀寫呢？所讀寫可以分成增加路由和減少路

由，`route` 的話意指 `route` 路由表的內容。在 Linux 上我們可以用 `netstat` 和 `route` 指令 `route` 路由表的內容並顯示之。它們是如何得到的呢？同時我們也可用 `route` 對路由表增加或減少其中的內容，這又是如何做到的呢？我們上為各位開這 `route` 的面。圖 1 為其運作的簡略示意圖。

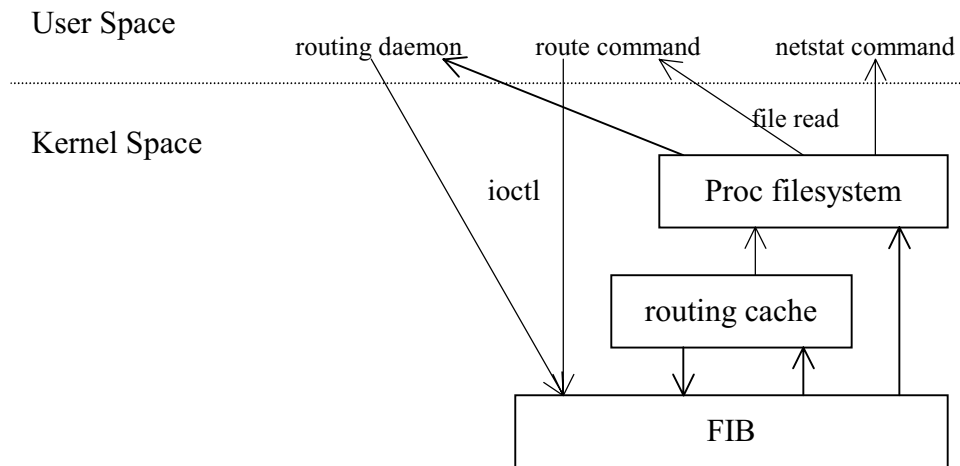


圖 1：路由表和 `route` 的運作

由圖 1 我們可以發現如果有人需要路由表中的資訊時，是經由 `proc file system` 來對其進行一般檔案的 `open` 和 `read` 的動作，這樣就可很輕易的得到相關的資訊。`FIB` 的資訊在 `/proc/net/route`，路由 `cache` 在 `/proc/net/route_cache` 中。而在 `proc file system` 中有 `FIB` 和路由 `cache` 所註冊的函式會將其中的資料放到相關的檔案中，這樣我們就可以經由 `file read` 檔案的動作取得相關訊息，而不需要直接接觸核心的值，這是不是既方便又安全呢？在最流行的 Linux distribution 中，如 redhat 6.0 中的 `route` 指令更可以用選項 `-C` 來印出路由 `cache` 的內容，如果沒有特別指明的話，`default` 會直接印出 `FIB` 中資料的內容，如同指明 `-F` 選項一般。

而如果我們需要進行增加或減少時則是經由 `ioctl` 請經由 `BSD socket` 介面來完成的，`BSD socket` 會傳送到 `INET` 層，這時便會檢視其請求的類型，如果是 `SIOCADDRT` 或是 `SIOCDELRT` 的便再交由 `ip_rt_ioctl` 處理，`ip_rt_ioctl` 會再根據請求類型分送到相關的函式處理。如果請求 `SIOCDELRT` 則先刪除在 `FIB` 中的資料項，再檢查路由 `cache` 中有沒有它的分身，有的話也一併刪除，至此完成其一。如果請求 `SIOCADDRT`

要增加路的話，則先檢查有沒這個介面的 fib_info，沒有的話則新增這個資料項，再新增有關這個目的位址的 fib_node，請注意此時並不會新增在路由表的資料項。但這一只有超級使用者才能進行這樣的動作，如果一般使用都可以用的話，那就不是下面大。例說明如 gated、route 和 netstat 等命令都這樣實作寫路由表和路由表的。

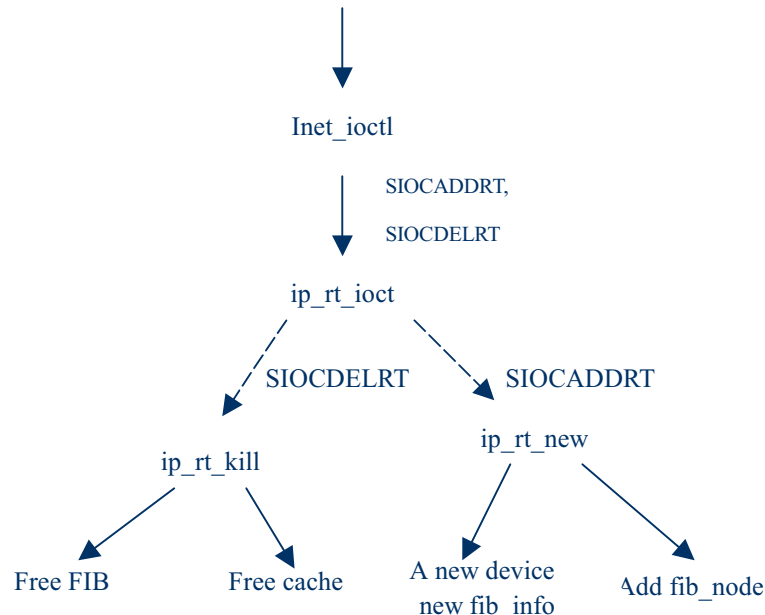


圖 1、IOCTL 與路由表

五、Benchmarking

我們先架好並設定一個 Linux 路由器。使用 RedHat 6.0、Kernel 2.2.5 版。PC 為 Pentium II 350 的處理器、64MB 的記憶體再加上兩個 Intel EtherExpress Pro 100 和兩個 R45 的線轉接頭。另外請注意一些 Kernel 中的選項、如 advance router 和 optimize as router not host 等，這些選項都可以讓你的路由更加快，例如有些網路卡可以接在卡與卡之間傳輸等，如果不了解的話請看一些說明。對網路設定好後就開始進行以下的過程，先 Kernel 中各模組花在轉送過程的時間，以了解到時間花在那和那是那個模組。最後再用 SmartBits 2000[13]來測這台 Linux 路由器的效能如何。

各別模組

這部份一開始是用 Kernel 所提供的函式 printk 加上 do_gettimeofday 做為工具來測

量，只能以 microsecond 做為單位。後來利用 x86 機器指令來量各模組的時間便可以達到以 nanosecond 為單位的目標了。不過請注意用 printk 來紀錄資料，如寫入資料到硬碟，可能會有不少問題存在，因為硬碟的 IO 總是較慢於資料的存取，如果你硬碟可用時間不大那可能很快地你的硬碟會掛掉。所以做這樣的測量在某些許的不確定性和誤差，不過多少還是可以做為參考。

microsecond 測量方法

我們可以使用一個較為粗略的方法，在我們要測量的函式或程式碼上下各用一個可以取得目前時間的函式，然後再把它們相減這樣就可以得到其差也就是執行時間了。在 Linux Kernel 中有個 do_gettimeofday 函式可以使用，功用和一般 unix 中所用的 gettimeofday 差不多，不過你如果想在 Kernel 中做這樣的事，如計算時間差那就非用這個函不可了。其宣告在 `linux/time.h`，實際程式碼在 `/usr/src/linux/arch/i386/kernel/time.c` 中，有不少 x86 組件部份。

nanosecond 測量方法

現今 x86 如 Pentium 等級以上的機器都有一個 RDTSC[14]機器指令可以做效能測量用。這個不只在 Intel 的 CPU 上有，AMD 等後來也有加入了，所以不用擔心了其它 CPU 的 CPU 這樣好用的東西就不能用了。RDTSC 的 counter 有 64-bit 長，每個 clock cycle 都加一，我們可以用 RDTSC 去讀這個 counter 的值，不過請注意的是讀出來的值是多少 cycle 而不是多少時間。在一台 500Mhz 的機器上，一個 cycle 就等於 2 nanoseconds，公式如 $\text{nanoseconds} = \text{cycles} / (\text{mhz} * 1e-3)$ 。當然用這個來量時間差對於一些個並不是對組件有經驗的人可能會感到困難，不過好在 `CC` 中可以找到範例接用，不用寫一組代碼在那加減乘除，不過也要加對地方，這樣就可以如同 do_gettimeofday 一般使用做為我們測量的工具。

測量的結果

我們在 load 不是很忙時分別對查詢 FIB 和路由表以及 ip_finish_output 和 ip_rcv

等各個函式，測量其在轉送過程時所花的時間，主要是想要了解 `ip_rcv` 所在之處。在 light load、64byte 資料時經過反覆測試，最後 `ip_rcv` 到所花的時間

`ip_route_output` —查路由 `ip_rcv`，`ip_rcv` 0.6 個 microsecond
`ip_route_output_slow`—查 FIB，`ip_rcv` 25 個 microsecond
`ip_finish_output`—`ip_rcv` 5 個 microsecond
`ip_rcv`—`ip_rcv` 8 個 microsecond

`ip_route_output` 和 `ip_route_output_slow` 我們在前面已經有提到過了，而 `ip_finish_output` 不論是自己機器或幫人轉送的封包 `ip_rcv` 會呼叫此一函式完成轉送，`ip_rcv` 一些資料，再把 skb [15]這個資料結構送給下層。而 `ip_rcv` 名詞意義就是收到封包後在 IP 層所要做的一些處理，先檢查標頭再看看是自己要收的還是要幫 `ip_rcv` 轉送的，再分別丟給負責處理的函式。因為 `ip_rcv` 做了許多比較所以花的時間比 `ip_finish_output` 多。`ip_rcv` 由上我們可知查詢路由 `ip_rcv` 的時間的 `ip_rcv` 是比查 FIB `ip_rcv` 多了，而查詢 FIB 或路由 `ip_rcv` 的比例跟網路的使用情形有關。最重要的是查詢路由 `ip_rcv` 的時間和其它函式比起來 `ip_rcv` 得 `ip_rcv` 多了，經由查詢 `ip_rcv` 每丟一個封包我們就節省了幾個 us 的時間，大費 `ip_rcv` 章還是有點代 `ip_rcv` 的。不過請 `ip_rcv` 意的是我們只對幾個主要的函式測量，所以它們加起來和總 `ip_rcv` 花的時間還是有段差 `ip_rcv`。在做測量時我們也不時用 top 來 `ip_rcv` `ip_rcv` load 的情形，最多也不 `ip_rcv` 過 2，`ip_rcv` Pentium 等級的處理器對於這樣的工作 `ip_rcv` `ip_rcv` 任。`ip_rcv` 是硬 `ip_rcv` 一 `ip_rcv` 在 IO 中，有點 `ip_rcv` 心會不會硬 `ip_rcv` `ip_rcv` 掉。最後如果你記得如我們前面所說 Kernel 有選一些可以加 `ip_rcv` 路由的選項那麼多少可以像我們一樣再多 `ip_rcv` 幾個 us。

介面至介面的延遲與輸出

我們用多 `ip_rcv` 測試 `ip_rcv` SmartBits 2000 在相同的環境下做測試。主要是用所 `ip_rcv` 的 Smart Application 這 `ip_rcv` `ip_rcv` 體測試。`ip_rcv` 手冊做好相關設定、設定測試路由器及設定所有的卡、如每 `ip_rcv` 卡的 IP、用的協定、MAC 位址等。最後我們測試其中三項：latency、throughput 及 packet loss rate。圖 `ip_rcv` 為們的實 `ip_rcv` 環境。

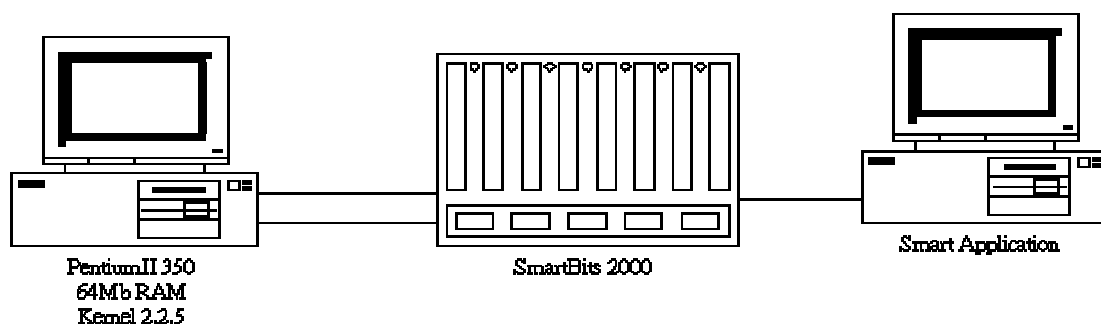


圖 1、用 SmartBits 測 Linux 路由器的環境

圖 1 為 throughput 的測試結果。一開始時 SmartBits 會全力來送封包。如果行不通，就用比較低的 rate 送，經由這樣反推的步驟進得出網路的能力是多少個封包/每秒。frame 越大則它的 throughput 也越大，已經接近這台 Intel EtherExpress Pro100 的網路能力，這也同時可以和後面的 packet loss rate 為 0。

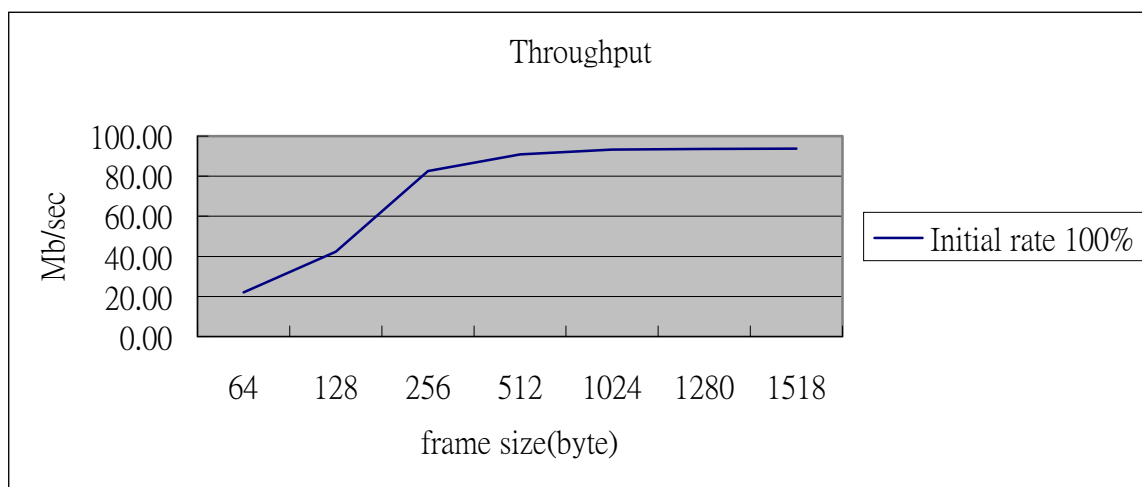


圖 1-1、Linux 路由器的最大 throughput

由圖 1-2 可知 frame 越大所花的時間越多，原因在於處理資料的時間，所以這部份的時間是線性增長的。另外一方面處理器所參與的時間，這部份的時間不會因 frame 的增大而有太大的變化。而 64 byte 的 frame 在 load 10 時花了少許時間，但 load 一到 40 就上不行了，相對的在 frame 大於 256byte 後這個情形就 reversed 了。請注意的是我們之前用 printl 的那幾個函式並不包含所有 Kernel 處理的時間，而且 Kernel 對此的處理時間

會因封包的多少而線變。

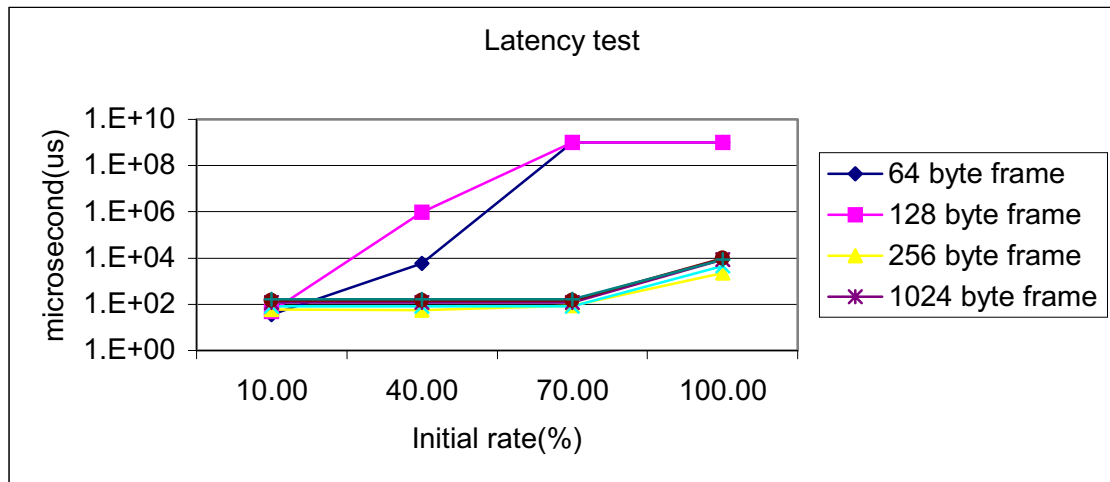


圖 二、Linux 路由器的 latency

圖 三 Packet loss 比較，可以做為前面之用。大可以發 512 byte 的線幾是不在的，因為幾沒丟掉什麼封包。相對來說在 64 byte frame 時，因單位時間處理的封包過多，因而不。

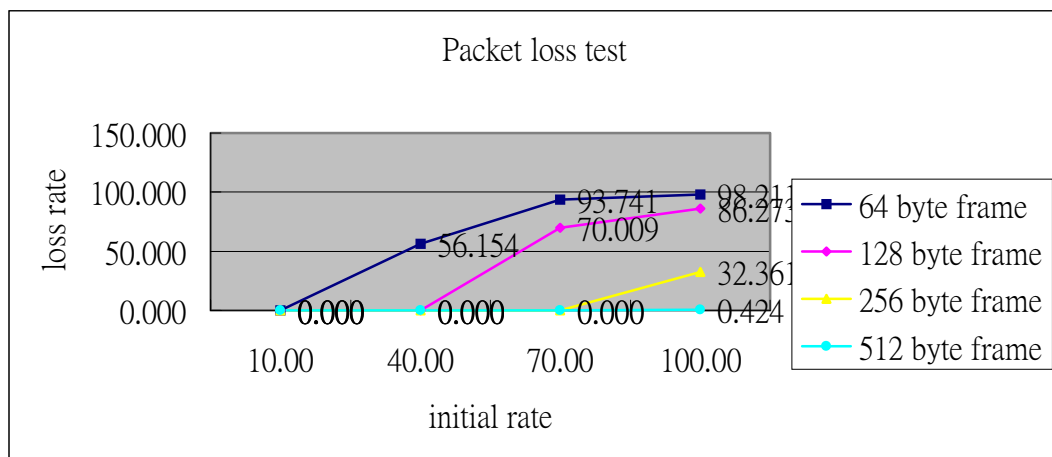


圖 三、Linux 路由器的 packet loss rate

六、結論

我們在前面為各位介紹了 Linux 的網路模組、路由並且測試 Linux 路由器。最後得到的結果是如果你用 Linux 做為路由器那在一個 load 不是很重的情形下，那它可能是一個讓你不用花大把的解決方案。由前面的結果得知路由的查詢在個封包轉送過程中並不是所在，相對的來說資料在記憶體和介面卡的通道如何加才是重點所在，不過 PC 的架構也只是著重在計算的能力上？做為路由器也不是它的使，

所以大端也不必太過端端，要不然那些端端網路設備的端端要端什麼過活。

七、參考資料

1. M Beck, H Böhme, M Dziadzka, U Kunitz, R Magnus, D Perworner, LINUX KERNEL INTERNALS Second Edition, Addison Wesley Longman, Inc., 1997.
2. David A Rusling, “The Linux Kernel”, 1996-1999, <http://metalab.unc.edu/LDP/LDP/tlk/tlk.html>
3. Transmeta Corporation, 1997-1998, <http://www.kernel.org/>
4. Brian Ward, “The Linux Kernel HOWTO”, 5 June 1999, <http://metalab.unc.edu/LDP/HOWTO/Kernel-HOWTO.html>
5. Alessandro Rubini, Linux Device Drivers, O’Reilly & Associates, Feb 1998.
6. Patrick Volkerding, Kevin Reichard, Linux in Plain English, ID Books Worldwide, Inc.
7. Nemeth Snyder, Seebass Hein, “UNIX System Administration Handbook”, Prentice Hall, Aug 1994.
8. Brecht Claerhout, <http://reptile.rug.ac.be/~rcoder/index.html>
9. Saravanan Radhakrishnan, “Linux – Advanced Networking Overview [version 1]”, 22 August 1999, <http://os.ittc.ukans.edu/howto.ps>
10. Douglas E. Comer, “Internetworking With TCP/IP [volI: Principles, Protocols, and Architecture, Second Edition]”, 1991.
11. Merit Network, Inc., 1999, <http://www.gated.org/>
12. Digital Magic Labs, Inc., 1999, <http://www.zebra.org/>
13. Netcom Systems, Inc., <http://www.netcomsystems.com/>
14. Intel Corporation, “Using the RDTSC Instruction for Performance Monitoring”, 1997, <http://www-cs.intel.com/drg/pentiumII/appnotes/RDTSCPM1.HTM>
15. Alan Cox, “Network Buffers And Memory Management”, September 1996, <http://metalab.unc.edu/LDP/>