封包經過 Linux 路由器的繞送分析

杜之雄 張舜理 林盈達
國立交通大學資訊工程系
September 2009

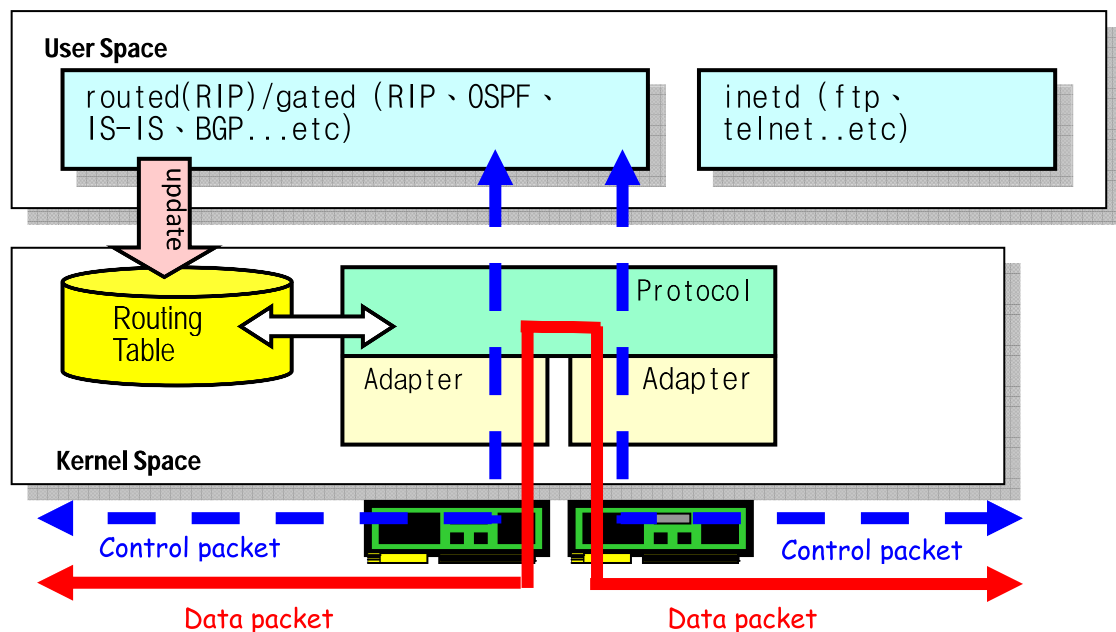
摘要

本篇文章探討封包經過 Linux 路由器的過程。一個封包從網路介面端口進入這台路由器起，經過核心的繞送處理後將封包從該路由器的網路介面端口送出，在這個過程中路由器的核心呼叫了哪些函式，函式當中哪些花.費較多時間？ 它在做甚麼動作？ 為什麼那些動作會花比較多的時間？ 針對這些問題，我們利用了 KFT[1]、printk 及透過 LXR[2]閱讀 Linux 原始碼，找出了 Linux 要作為路由器這個角色所需要使用到的相關函式，在函式的頭尾處插入 rdtscll 來擷取函式執行的時間戳記，蒐集資料後再來整理分析。依封包繞送所通過的協定堆疊(protocol stacks)與網卡的 DMA 收送程序，將測量過程切分成六階段的時間。量測發現在我們實驗環境下，一個 ICMP 封包從進入路由器到離開的平均時間為 29.12μs，最耗時部分是在資料鏈結層的封包接收處理，占總時間的 37%，其次是鏈結層的封包傳送處理占 27%，其他分別為網路層的封包接收處理占 25%、網路層的封包傳送處理占 5%、網卡的 DMA 傳送和接收都在 3%左右。

關鍵字：路由器，Linux，KFT，rdtscll，延遲分析

一、簡介

首先我們介紹 Linux 路由器運作的基本觀念



圖一 Linux 路由流程及模組[3]

由圖一所示，底下兩個模組代表網路卡，是封包進出路由器的端口(port)，路由器本身其實就像是一台有著多個網路卡的 PC，它的作業系統就嵌在機器的

ROM 裏頭，而路由器上的 ROM 就相當於 PC 上的硬碟。開機後自動會載入網卡的驅動程式(Driver)和相關路由器運作所需要的服務(Daemon)。路由器除了本身開機時載入的小部分靜態路由表(Static Routing Table)之外還可以透過控制封包(Control packet，圖一虛線資料流部分)跟其它在網路上的路由器交換路由資訊，動態計算產生動態路由表(Dynamic Routing Table)，圖一實線資料流部分則表示一般路由器大部分時間所做的事情--幫忙繞送來自四面八方的資料封包。我們在這篇文章中僅會探討 Data plane 部分的網路封包在路由器中運作的情形。

接著在第二節中會對我們的量測環境和量測方法做介紹，包括在量測過程中需要安裝哪些可以觀察 Linux 核心運作的工具，以及如何藉由插 code 的方法來記錄封包在哪些函式中所花的時間。在第三節我們會對第二節中所產生的結果來做分析比較並討論封包在核心裡頭被哪些函式處理過，那些函式做了哪些動作。最後我們會將這些分析比較後的結果歸納在最後一節結論的部分。

二、追蹤量測封包在 Linux 路由器運作路徑的方法

量測環境

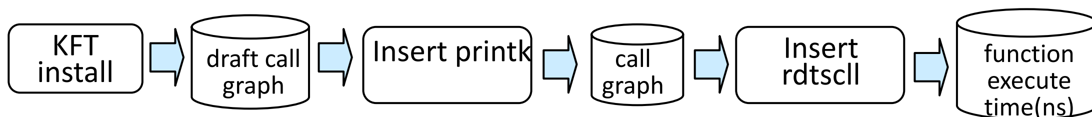
圖二 測量環境示意圖

如圖二所示 Host C 爲模擬 Linux 路由器的待測物，它需要至少兩張網卡分別設定成兩個不同的網段。本文範例中用 10.10.10.0/24 及 192.168.1.0/24 兩個不同的 class C 網段，爲了使不同網段的封包在經過 Host C 這個路由器時，Host C 的核心可以幫忙繞送，我們必需打開 Host C 上的 ip_forward 功能。實驗過程中爲了方便每次重新開機時 ip_forward 的功能會被自動開啓，在/etc/sysctl.conf 檔案裏的"net.ipv4.ip_forward"，設定值由 0 改成 1。此外爲了讓 Host C 環境變得較單純，把它上面除了與路由器運作相關及讓系統運作的基本服務外，其他如 firewall、藍芽、selinux、auditd..等服務關掉。Host A 和 Host B 則爲任意兩台可以執行 ping 及 ttcp 指令的機器，IP 要分別設定成相對於 host C 兩個網段的 IP 地址，在此例分別爲 Host A:10.10.10.1、Host B:192.168.1.1 並將預設網路閘道指向 Host C 的兩個不同網卡的 IP 地址，分別爲 10.10.10.254、 192.68.1.254。設定好了之後找較短的網路線如圖二的接線方式接起來。可由 Host A 下 ping 192.168.1.1 測試是否能將封包經由 Host C 送達 Host B 來驗證是否基礎環境已架設成功。測試環境詳細的硬軟體規格，請參考表一。

表一 測量環境元件

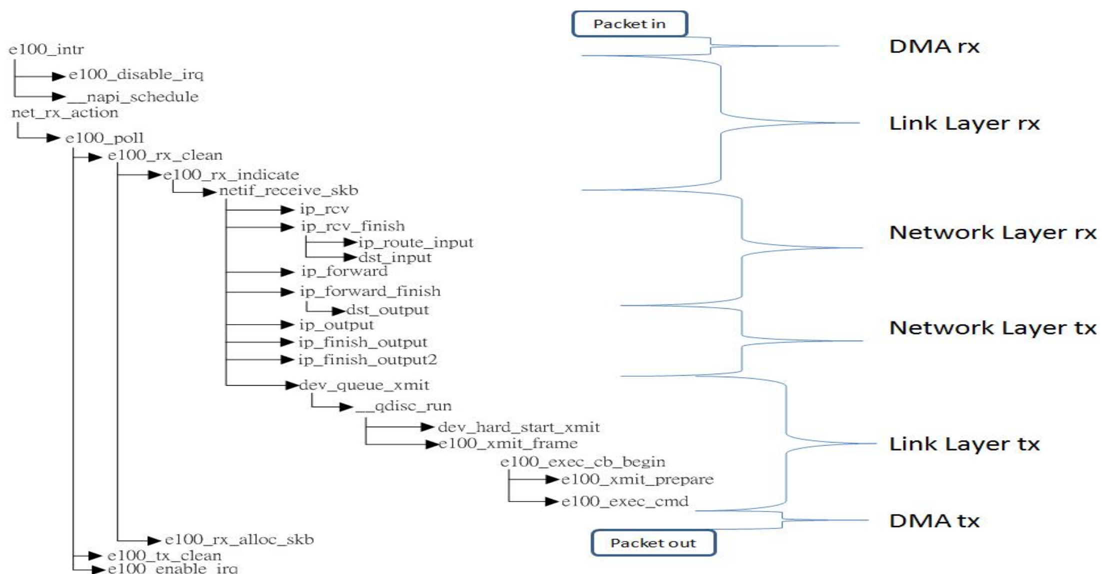
基本硬體	3PCs,4NICs
Linux 路由器 CPU	AMD Athlon 1.1GHz
作業系統	CentOS5.3(kernel 2.6.28)
核心函式呼叫分析工具	KFT for 2.6.28
可插入核心中的時間分析函式	rdtscll, printk
封包產生工具	ping, ttcp
路由器的網卡型號/驅動程式	Intel pro100/e100
測試封包大小	一個含 56 bytes data 的 ICMP 封包

測量方法 1-不包含 DMA 時間



圖三 追蹤及測量的簡化流程圖

Linux 核心函式是非常龐大且複雜的;試想 Linux 核心呼叫哪些函式來處理一個封包通過路由器過程的所有動作,以及這些函式分別花了多少時間? 針對此問題，我們找到相關的分析工具，整理成圖三所示的測量方法流程一步步來得到我們想要的詳細數據，首先要找到和機器上 Linux 核心版本相同的 KFT 版本，藉由用 patch 的方式植入這台 Linux 路由器的核心，再透過一些簡單的操作(參閱 KFT 網址[1])，獲得系統核心在某段時間內呼叫到的所有核心函式，使用 KFT 提供的 scripts--kd 整理出函式間的呼叫順序。附帶一提，KFT 上有每個函式所執行的時間資訊；但因植入 KFT 後的作業系統會有 30%~200%的額外負荷(overhead)，所以不採納這項結果，只取用其函式間 call graph 圖的部分，另外可以再使用不會造成系統負擔的其他方法來獲取更詳細的時間資訊。



圖四 系統處理封包的函式呼叫關係圖

從 KFT 的結果中我們雖然可以知道函式的呼叫順序,但無法知道函式間的父

子關係，所以下一步我們拿 KFT 的 call graph 結果加上利用 LXR[2]去閱讀相關的原始核心程式後，我們可以找出函式的位置，這樣就可以在相關的函式頭尾插入 printk 來驗證核心處理封包時詳細的呼叫流程父子關係(圖三 Insert printk 到產生 call graph)，經過漫長的插 printk 及 rebuild kernel 的過程後，最後我們可以整理得到我們需要的函式間完整呼叫關係圖(圖四)，關係圖內容部分我們會在下一節來詳細說明。得到呼叫流程圖後，我們便可以在原來插入 printk 的地方用 rdtscll 來取代，用 rdtscll 來抓取系統的 counter 值，最後用 printk 在封包離開路由器後，下一個封包進來前的函式(作者選擇了 net_rx_action()尾端)印出那些插入 rdtscll 函式的頭尾相減後的 counter 數，最後再依其函式間的父子關係圖將其子函式所花的時間扣除，就是函式扣除子函式後真正的執行時間。

關於系統其他的 Interrupt 是否會影響測試數據的問題在 e100 的驅動程式中有做了一些處理，當網卡將資料複製到 DMA 後會對系統產生一個 Interrupt，當 e100_intr()這個 driver 函式(參考圖四中的第一個呼叫的函式)確認這個 Interrupt 是由網卡發出後，會呼叫 e100_disable_irq()來占有 CPU 資源不讓其它的 Interrupt 發生，直到 e100_poll()處理完 skb 中的資料在尾端呼叫 e100_enable_irq()(圖四中最後一個被呼叫的函式)。

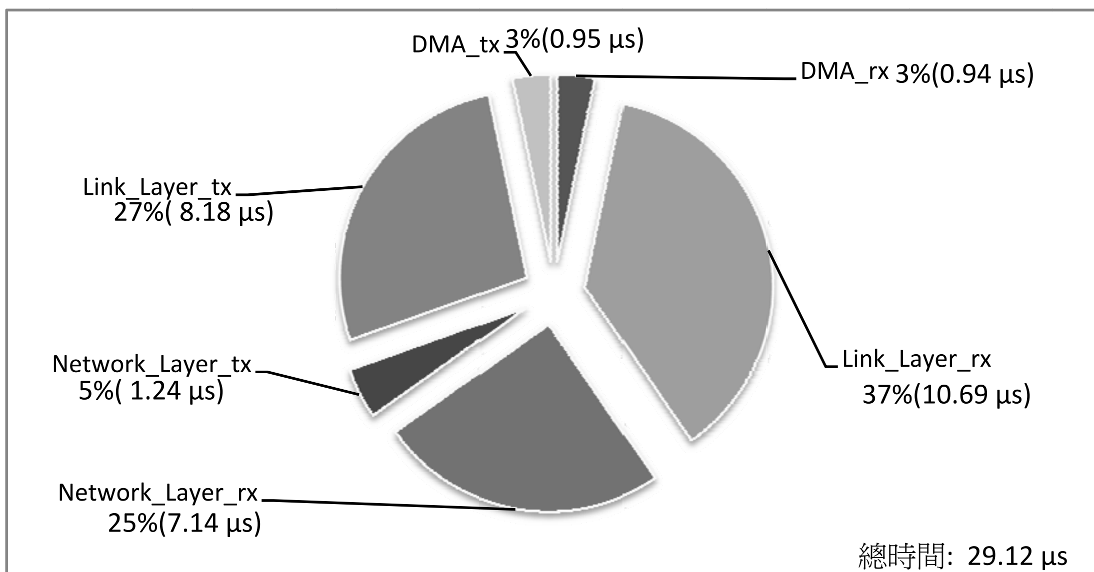
測量方法 2-封包接收及傳送的 DMA 時間

關於封包的接收，由於當 Linux 系統知道有封包到達的時間點(e100_intr()被呼叫的時間點)，已經是網路卡將封包 copy 到 DMA 後的時間了，所以我們很難直接在某個函式的頭尾插入 rdtscll 來獲取 DMA 的時間數據。封包傳送的部分，讀過 e100 driver 原始碼我們會知道，系統傳送封包是直接將原來 sk_buffer 的指標直接指到 DMA 的位址，之後資料要從 DMA 複製到網卡這段時間是由硬體自動完成，我們一樣無法精確在核心中特定函式頭尾插 rdtscll 來獲取所花費的時間。關於這點我們想到在 Host C 上用 ttcp 連續打封包到 Host A，假設連續兩個封包的間隔時間短到我們可以忽略，配合讀 e100 的 driver 內容知道當系統在接收封包時，系統會先把原來網路卡記憶體跟系統實體記憶體做 mapping 的空間先 unmap，再把這個實體記憶體的空間指標交給 sk_buffer，完成所有原來 DMA 中的資料用指標的方式交給 sk_buffer 後，系統會再重新分配一個新的實體記憶體空間來跟網卡的記憶體做 mapping，成爲新的 DMA 空間。我們假設 DMA 一建立好的瞬間就會馬上去做下一個將網卡上的資料複製到系統實體記憶體的動作，所以我們在 e100_rx_alloc_skb()裏的 pci_map_single()函式之後插入 rdtscll 作爲紀錄的起始點，再到下一個 e100_intr()被呼叫的開頭處插入 rdtscll 作爲紀錄的終點。將這兩個 rdtscll 得到到時間相減再除以每次 DMA 處理期間的封包量，便得到每單位封包花在 DMA 接收上的時間。關於封包傳送的 DMA 時間(資料從實體記憶體複製到網卡記憶體的時間)也是用同樣的方法在 e100_xmit_prepare()中的 pci_map_single()後和 e100_tx_clean()中的 pci_unmap_single()前插入 rdtscll，最後將這兩個 rdtscll 的值相減後得到。

三、封包處理流程分析

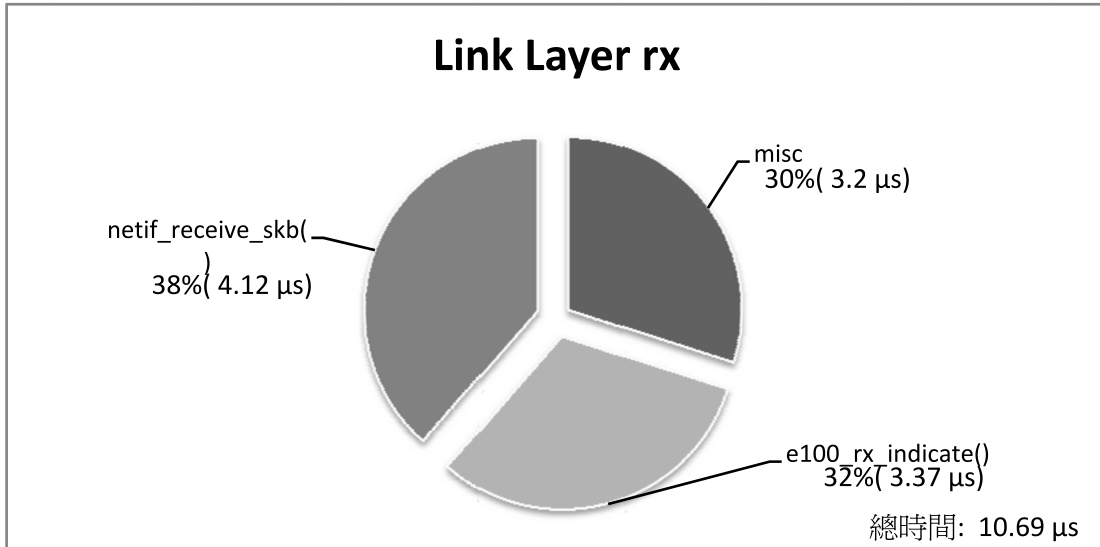
由上節內容中透過 KFT 及 printk 方法得到的關係圖如圖四所示，我們可以將封包經過路由器的過程由進入到出去的順序分成六個時期。

1. 網卡 DMA 接收(DMA rx):從封包到達網卡開始直到網卡把資料複製到 DMA 記憶體結束。
2. 資料鏈結層接收(Linker Layer rx):從網卡對作業系統發出 interrupt 藉由 driver 函式 e100_intr()被觸發開始直到呼叫 netif_receive_skb()函式把訊框副本提供給每個 L3 已註冊的協定處理常式，準備把 sk_buff 資料往網路層傳送的這段期間。
3. 網路層接收(Network Layer rx):從被 Linker Layer 的 dev_add_pack ( )觸發 ip_rcv()開始到 ip_forward_finish()函式結束。
4. 網路層傳送(Network layer tx):從函式 ip_forward_finish()的尾端將 dst_output() return 回去開始到 ip_finish_output2()結束。
5. 資料鏈結層傳送(Linker Layer tx):從 dev_queue_xmit()函式被呼叫開始到 e100_xmit_prepare()中呼叫 pci_map_single()來 mapping 要傳送的 sk_buff 到 DMA 的記憶體位址準備將資料送到網路卡作爲結束。
6. 網卡 DMA 傳送(DMA tx):從封包指向 DMA 位址開始將資料複製到網卡記憶體完畢作爲結束。



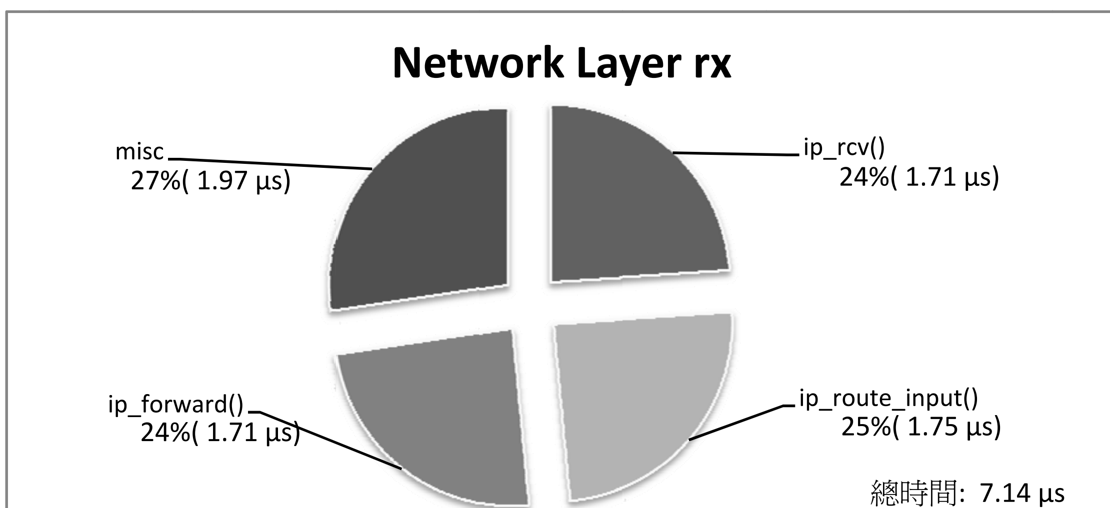
圖五 一個 56 bytes 的 ICMP 封包經過路由器的時間比例分析圖

圖五爲一個 56 bytes 的 ICMP 封包經過路由器的時間比例分析圖，我們發現在 Link Layer 處理接收封包的時間是最長的占所有時間的 37%，其次是在 Network Layer 處理封包的接收 25%，傳送部分在 Link Layer 的 18%大於在 Network Layer 的 14%，DMA 收和送的部分在 e100 網卡的表現上幾乎相同。光是接收封包的過程 DMA_rx + Link_Layer_rx + Network_Layer_rx 就占了總時間的 66%。接下來分別對 Link Layer 及 Network Layer 最花時間的兩個階段來看最花時間的是哪些函式，再把這些函式的功能做個簡單的介紹以理解爲何它需要花較多的時間。



圖六 Link Layer rx 階段中函式的執行時間比例分析圖

由圖六可知 Link Layer rx 部分光是 e100_rx_indicate()和 netif_receive_skb()就占了 70%。e100_rx_indicate()被呼叫後他會先檢查 cb_complete 狀態位以確定 DMA_rx 中的資料是否準備好了,再呼叫 pci_dma_sync_single_cpu 來讓 CPU 在取消 DMA mapping 前，具備讀取 DMA 空間的能力，接著就是 unmap DMA，此外還需要呼叫 skb_reserve()、skb_put()、eth_type_trans()等函式來將 sk_buffer 的各 layer 的 header 和 data 位址對齊,並設定此封包的類型及協定。netif_receive_skb()不僅是 Link Layer rx 階段最耗時的函式同時也是所有封包經過路由器流程中最耗時的函式，它的主要任務有三項，第一是把訊框的副本傳遞給每個協定，第二是把訊框的副本傳遞給被關聯到 skb->protocol 的 Network Layer 協定處理函式(在本編文章是指 ip_rcv()這個函式),第三是要搞定 Link Layer 中必須處理的橋接功能，包含封包 source 和 destination 的 MAC 都要記錄到 neighbour table 中。



圖七 Network Layer rx 階段中函式的執行時間比例分析圖

由圖七可知 ip_rcv()、ip_route_input()ip_forward()這三個函式就占了 Network Layer rx 階段的 73%。其花費時間排名順序爲 ip_route_input() > ip_forward() >

ip_rcv()。首先來瞭解最花時間的 ip_route_input()。它的主要工作是決定封包之後的命運。ip_route_input()會去看 IP 的 destination，如果是自己就會呼叫 ip_local_deliver()往 TCP 層送，如果是其它的 IP 位址，就到 cache 上的動態路由表中查看是否有命中的 routing 資訊，如果沒有命中還會再另外呼叫 ip_route_input_slow()或 ip_route_input_mc()來查找靜態路由表，如果都找不到就會將封包丟棄。ip_forward()工作會檢查是否有 IPSEC 的方針需要遵守，是否有設定 Router Alert 選項，檢查 ICMP 的 TTL 是否小於 1，對 TTL 值做遞減，決定是否要發送 ICMP redirect 封包等等。ip_rcv()的主要工作在對傳進來的封包做基本的健全度檢查，例如對 Header 和 data 做長度檢查和封包 cksum 檢查，最後還要確定 sk_buff 的長度是否大於或等於 IP Header 所回報的長度，以及確定封包的總尺寸大於等於 IP Header 的長度。如有任何錯誤即會把封包丟棄。最後再呼叫 ip_rcv_finish()。

四、結論

我們在前面爲各位介紹了 Linux 路由器的模組的基本觀念，並了解如果想知道一個封包通過路由器的過程，可利用兩張網卡加上裝有 Linux 作業系統的 PC 來模擬路由器，另外再找兩台電腦連上此路由器成爲一個簡單的測試環境。運用 KFT、printk、rdtscll、ping、ttcp 等工具得到路由器在處理封包時所呼叫到核心函式的 call graph 以及函式所執行的時間。接著將此過程分成六個不同的階層來分析所花費的時間，結果知道當一個 data 大小爲 56 bytes 的 ICMP 封包經過路由器的平均時間爲 29.12 μs，其中最耗時的部分爲 Link Layer rx ，占總時間的 37%，其次是 Link Layer tx 的 27%，其餘的分別爲 Network Layer rx 的 25%、Network Layer tx 的 14%、DMA 傳送 3%和接收 3%。配合閱讀相關介紹 Linux 網路核心和 Linux Driver 的書籍[4][5]後，我們瞭解爲何資料在接收的時會比傳送花較多的時間，主要因爲接收封包過程中會先讀取封包中 Header 所包含的許多資訊，並將此封包之後傳送所要用到的資訊放進 sk_buff 中相關的 structure 欄位；例如封包要送往的目的端資訊在接收階段就已在相關的 routing table 或 neighbor table 中填入或查詢。而在傳送時所呼叫的函式 ip_outout()、ip_finish_output()..等只需要做少部分的檢查及是否要切封包等處理，不需再花時間查詢一次，所以接收封包明顯較傳送時耗費時間。

參考資料

[1]KFT(Kernel Function Trace),http://elinux.org/Kernel_Function_Trace/.

[2]LXR, http://lxr.linux.no/.

[3]楊佳欣,林盈達, ”追蹤與測試 Linux 路由器,” 網路通訊 110 期, Nov 1999.

[4]Christian Benvenuti, ”Understanding Linux Network Internals,” O’Reily Media, Dec 2005.

[5]Jonathan Corbet, ”Linux Device Driver 3rd Edition,” O’Reily Media, Feb 2005.