Socket 層的中斷與記憶體複製分析

賴育聖 張舜理 林盈達
國立交通大學資訊工程系
October 14, 2009

摘要

本文介紹一種方法來分析 Linux 在 Transport Layer(傳輸層)中傳輸函式的執行流程並量測函式執行所花費的時間,藉此找出傳輸 bottleneck(瓶頸)所在並加以分析。傳輸層的瓶頸主要發生在 system call(系統呼叫)與資料的 memory copy(記憶體複製)處理過程(user space 與 kernel space 之間)。以 TCP 接收函式為例子,應用程式呼叫 read()函式請求 kernel(核心)處理接收的動作,在 Linux2.4 以前的版本是藉由軟體中斷來處理系統呼叫,在 2.6 以後的版本開始使用 CPU 快速呼叫指令 sysenter/sysexit 來處理。雖然在新版的核心使用這組指令已改善系統處理系統呼叫的效能,但仍須花費約 2.39 μs 的時間,占用整體接收函式所花時間的 13.13%。而 tcp_data_queue 跟 tcp_recvmsg 兩個函式則因為將接收到的資料存入 sk_receive_queue 後,再從 queue 中取出資料來完成資料 kernel space 到 user space 的記憶體複製過程而花費約 4.6 μs 的時間,占整體時間的 25.19%。因此可嘗試針對系統呼叫與記憶體複製的處理過程來改善,可獲得較佳的執行效率。

關鍵字：Linux , 傳輸層, Socket, 中斷, 記憶體複製

一. 簡介

Linux 在網路傳輸的應用可謂多如繁星,一般的網路傳輸服務都會使用傳輸層[1]的 socket interface[2],所以本文針對 socket interface 做一次完整的執行過程追蹤與量測函式執行所花費的時間來找出效能的瓶頸處。

Socket interface 可分為 Stream socket、Datagram socket 和 Raw socket,其中 Stream socket 是使用 TCP,在建立的過程中需要指定 server 的 IP 位址和 port 來建立連線,因此又稱為 connection-oriented socket。而 Datagram socket 是使用 UDP,不需提供任何資訊來建立連線,它只需在發送的封包中包含目標處的 IP 位址即可,故又稱為 connectionless socket。最後,Raw socket 最大的特點是在它允許使用者自訂封包的標頭,因此可利用這特性來實現一些應用,較著名的像是 ICMP(Linux 上著名的網路工具-traceroute 就可支援使用 ICMP echo 的封包來追蹤封包路徑)、和 OSPF(一種使用 Dijkstra 演算法計算到目的地最短路徑的鏈結狀態路由協定)等。

我們將在底下文章依序介紹量測過程的實驗環境和方法,以及實驗量測的結果,並根據結果加以分析和做結論。

二. 實驗環境和方法

我們使用 Unix/Linux 系統上著名測試網路承載流量的工具-ettcp[3]所發出的 TCP 和 UDP 封包來當作流量，以及一般系統皆有內建的 traceroute 所發出的 ICMP 封包當作 RAW 的流量。再藉由重編譯系統核心開啟 Kernel function trace(KFT)[4] 這套工具的支援來擷取核心中 TCP、UDP 和 RAW 三種協定在傳收流量時呼叫過哪些函式，但由於擷取出的內容包含非傳輸層的其它呼叫函式，因此對照他人的 call graph[5][6][7]來排除非傳輸層的函式，並驗證函式的執行流程來得到如圖 1, 2 的 call graph。

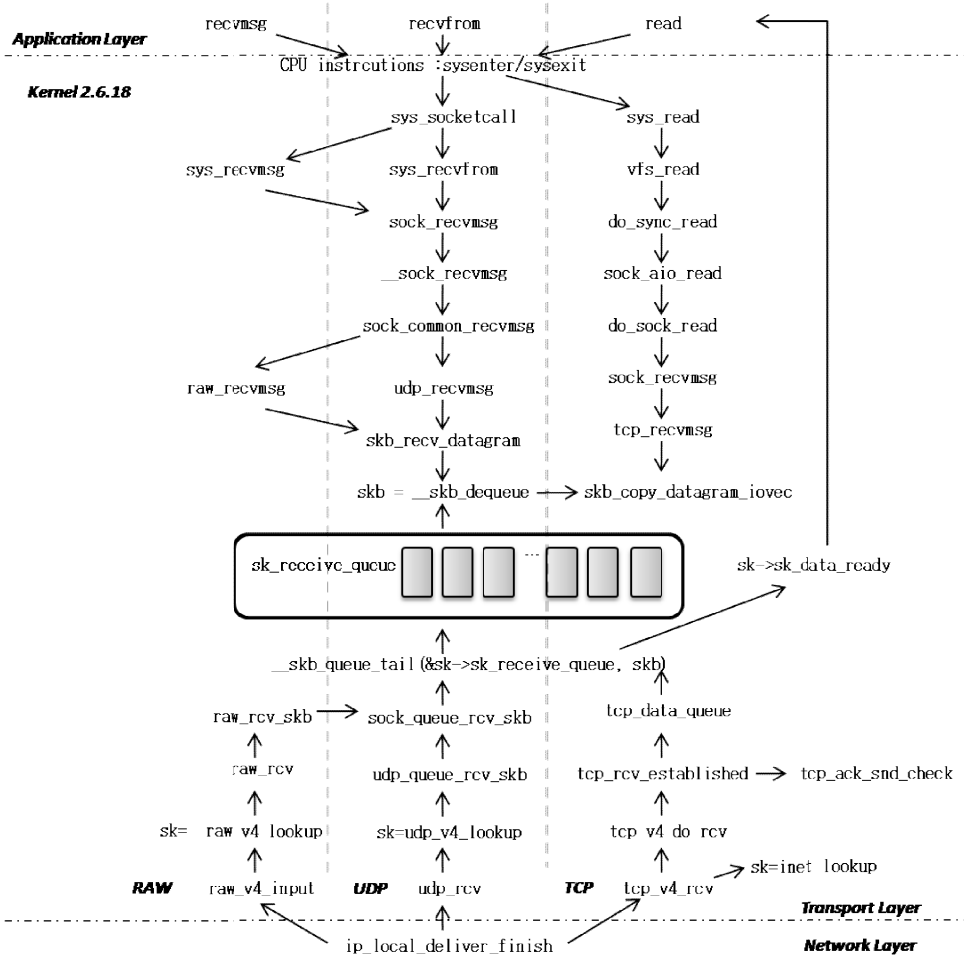


圖 1 接收端的 call graph

根據圖 1 可知三種協定在接收端的運作過程。首先應用層程式使用系統呼叫後 (recvmsg、recvfrom 和 read)，核心會一路呼叫數個函式至 skb_recv_datagram(UDP, RAW)和 tcp_recvmsg(TCP)來等待封包的送達。而當封包經由 IP 層往上送時，會在 ip_local_deliver_finish 函式中根據其協定來決定各自的傳輸層接收函式(raw_v4_input、udp_rcv 和 tcp_v4_rcv)。在經過數個函式處理後，會呼叫__skb_queue_tail 函式將封包中的資料存入 sk_receive_queue，然後呼叫 sk_data_ready 來通知在等待的函式(skb_recv_datagram 和 tcp_recvmsg)可以開始從 sk_receive_queue 提取資料。當函式提取資料後，會開始將資料從 kernel space

複製到 user space(raw_recvmsg、udp_recvmsg 和 tcp_recvmsg)，並藉由函式一路的向上結束回傳，讓應用層的程式可接收到資料，完成整個接收端的工作。

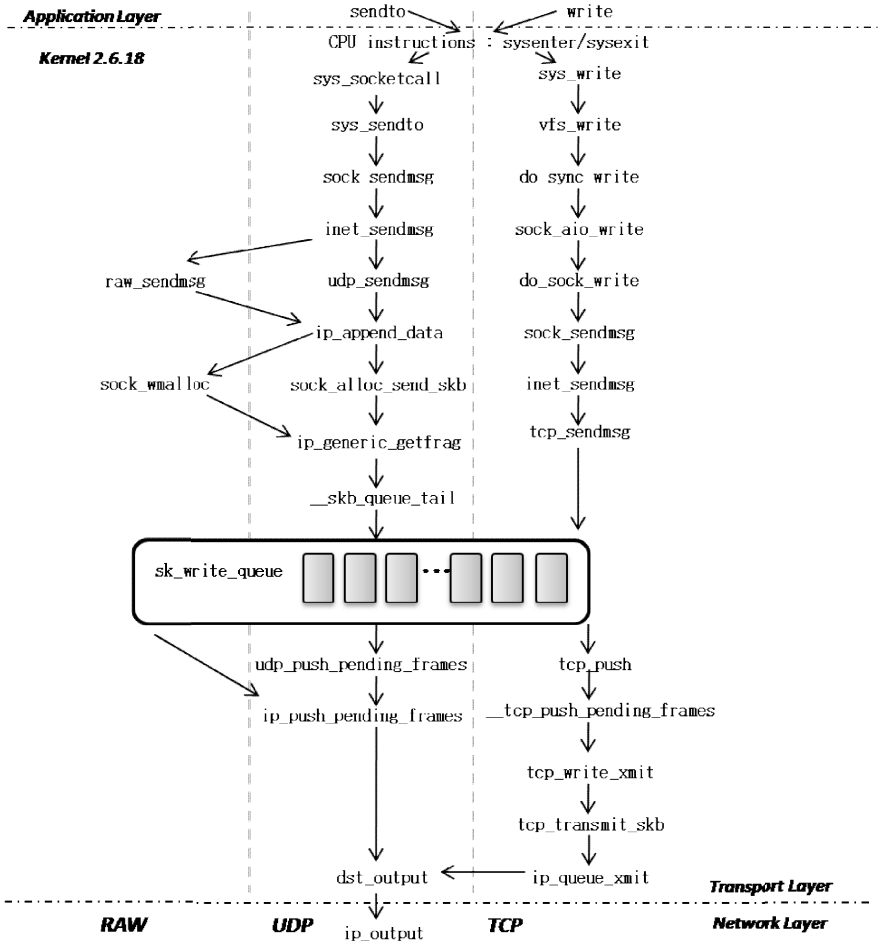


圖 2 傳送端的 call graph

圖 2 則是應用層程式使用系統呼叫後，一路經過不同的函式將資料從 user space 複製到 kernel space(ip_append_data 和 tcp_sendmsg)，再將資料放入 sk_write_queue，之後 udp_sendmsg 和 tcp_sendmsg 分別呼叫 udp_push_pending_frames 和__tcp_push_pending_frames 來提取資料(raw_sendmsg 直接呼叫 ip_push_pending_frames)，然後再一路往下呼叫其它函式將資料送往 IP 層。

因此根據 call graph 所呈現的函數，分別進行插入 rdtscll()[8]的動作，我們藉由 rdtscll()函式可拿到 TSC 暫存器值(CPU clock 數)的功能，將 rdtscll()插入目標函式的前後，再經由底下公式換算即可得到花費的時間。

Spend time=(end_cycle-begin_cycle)/CPU frequency

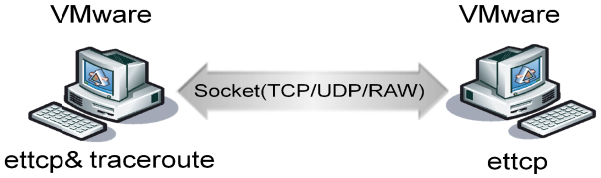
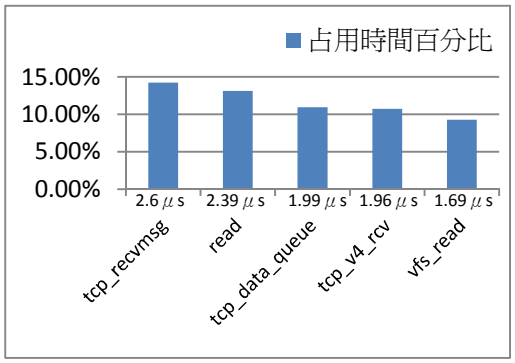


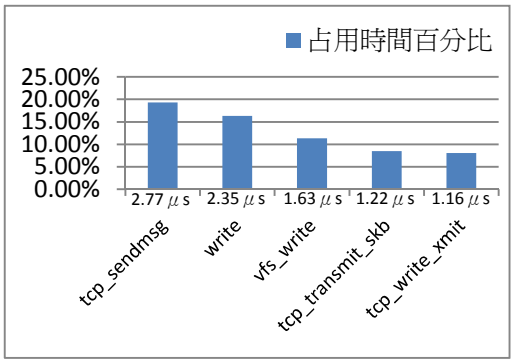
圖 3 實驗環境圖

當目標函式都插入 rdtscll()函式後，建立如圖 3 的實驗環境。我們使用 VMware 模擬兩台電腦簡單的對接，再使用 ettcp 和 traceroute 產生三種協定的流量，這時就可以得到目標函式所花費的時間，詳細實驗結果如圖 4, 5, 6。

三. 量測結果(僅列出占較大比例的函式)

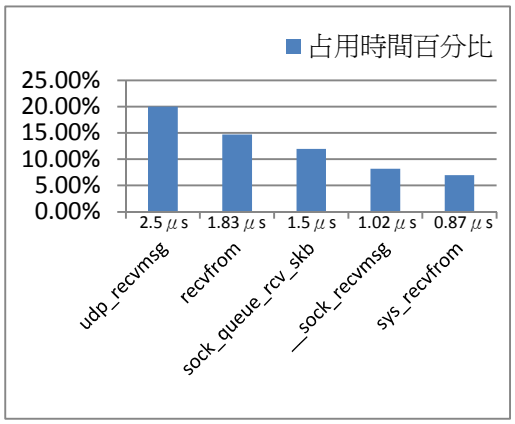


(a)TCP read 函式所占時間與比例

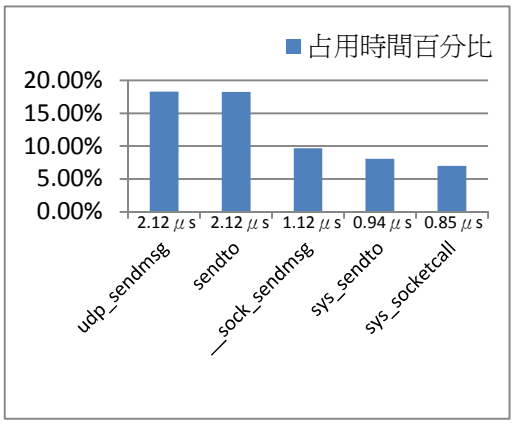


(b)TCP write 函式所占時間與比例

圖 4 TCP 函式收送所占時間與比例

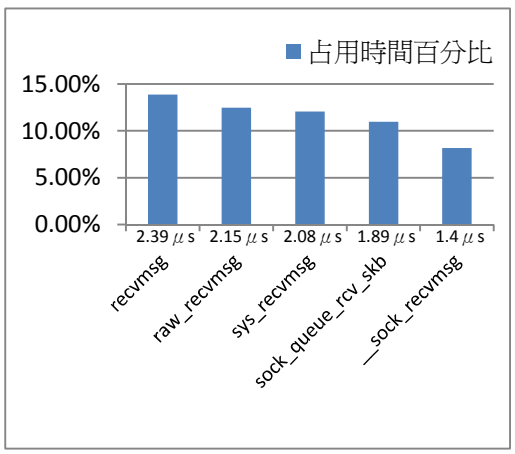


(a)UDP read 函式所占時間與比例

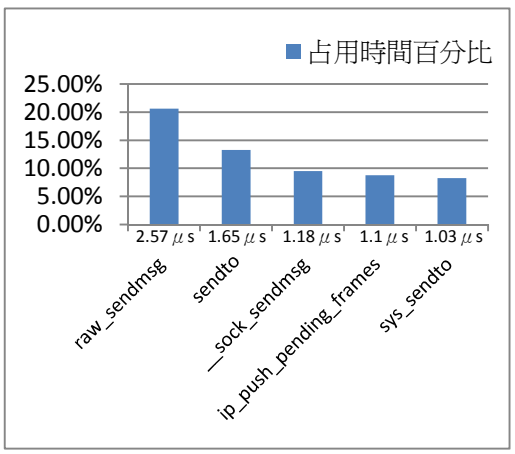


(b)UDP write 函式所占時間與比例

圖 5 UDP 函式收送所占時間與比例



(a)RAW read 函式所占時間與比例



(b)RAW write 函式所占時間與比例

圖 6 RAW 函式收送所占時間與比例

圖 4(a)的 TCP 接收(read)花最多時間的是 tcp_recvmsg，其次為 read，而圖 4(b)的 TCP 傳送(write)花最多時間的是 tcp_sendmsg，其次為 write。圖 5 的 UDP 花最多時間的前兩名函式為接收時的 udp_recvmsg 和 recvfrom 與傳送時的 udp_sendmsg 和 sendto，而圖六的 RAW 則為接收時的 raw_recvmsg 和 recvmsg 與傳送時的 raw_sendmsg 和 sendto。

四. 結果分析

因此根據數據的呈現，我們知道花較多時間處理的函式可以根據其功能大致分為系統呼叫和記憶體複製兩種來分析，其它少數幾個不在這兩種裡面的函式也會在後面分析。

1. 系統呼叫

系統呼叫在 Linux 2.4 以前的版本，是使用軟體中斷(int 0x80)通知作業系統將 user mode 狀態轉換為 kernel mode 狀態，並根據系統呼叫代碼得知系統核心該提供呼叫者什麼服務。但在 Linux 2.6 之後的版本改為使用 Intel 所提供的 CPU 快速呼叫指令(sysenter/sysexit)[9]來完成系統呼叫的模式切換，這組指令由於比起舊有的軟體中斷(int 0x80)省去了一些權限檢查和 push-pop 的動作，所以執行的效率會比軟體中斷更佳(根據[9] p. 22 可知道效能改善約 70%)。雖然效能因這組指令已有了大幅度的改善，但扣除掉中斷所花的時間後，整個系統呼叫的運作流程仍包含像是模式轉換、跳躍去執行目標函式等的時間。從圖 4~6 中的 read、write、recvfrom、sendto 和 recvmsg 函式需花費 1.5~2.5 μs 的執行時間(占整體時間的 13.13%)可知道系統呼叫仍需花費一段不短的時間。

2. 記憶體複製

圖 4~6 中除了系統呼叫函式外，佔用超過 10%時間比例的函式大多都是負責資料位址或資料內容從 user space 轉換至 kernel space 或反之從 kernel space 轉換至 user space 的記憶體複製動作，這些動作會因為配置資料指標、配置新的記憶體空間、釋放舊的記憶體空間以及做複製的動作而佔用較大量的時間，像是 tcp_recvmsg(14.25%)、tcpsendmsg(19.3%)、udp_recvmsg(20.02%)、raw_recvmsg (13.87%)、vfs_write(11.35%)、sys_recvmsg(12.08%)和 tcp_data_queue(10.94%)。

3. 檢查封包內容

sock_queue_rcv_skb 和 tcp_v4_rcv 函式花較多時間(分別為 1.89 μs 和 1.96 μs)在呼叫 sk_filter 來檢查封包的協定、狀態以及來源網址等資料。而 udp_sendmsg 和 raw_sendmsg 則是花時間(分別為 2.12 μs 和 2.57 μs)在呼叫 ip_route_output_flow 來得到封包 route 的路徑。

4. ICMP

RAW 協定下的接收端，共有兩個函式因為 ICMP 封包的 request/reply 特性，而有花較多時間處理的現象。ip_local_deliver_finish 會因呼叫 icmp_rcv 的函式來處理 ICMP 協定，造成量測數據包含非單純 RAW 協定的運作時間。另一個是 sk_data_ready，這個函式會花更多的時間(約 1.7 μs)從 wait_queue 中尋找對應的接收函式，並將其呼叫起來運作。這點 TCP 和 UDP 並不會發生，原因在於 TCP 和 UDP 的封包傳送在本文簡單的對接實驗環境下，不會有任何的等待或延遲導致接收函式因等待了一段時間而被放置於 wait_queue，然而 RAW 會因為 ICMP 等待對方 reply 封包的特性讓接收函式因等待而被放入 wait_queue，造成上述的現象。所以我們已經將以上述的兩個因 ICMP 協定而多出來的時間從數據中排除。

五. 結論

在傳輸層中，以系統呼叫及記憶體複製為較嚴重的瓶頸，對於 Linux 網路程式、服務的開發者而言，若能夠對於網路傳輸過程有瓶頸的函式有所了解，可以幫助他們釐清一些可能因為瓶頸而造成的效能問題，也可以建立正確的思路來解決效能瓶頸的問題。當然也相信在未來為了因應更注重效率的網路傳輸服務，會有人提出好比 sysenter/sysexit 這類在硬體或軟體上改善瓶頸的解決方案，讓網路傳輸的效能更加完善。

參考資料

[1] Andrew S. Tanenbaum, “The transport layer,” in Computer Networks, 4th ed. Prentice Hall, 2002, pp. 481-555.

[2] Socket interface, “Internet socket,” 24 September 2009, http://en.wikipedia.org/wiki/Internet_socket.

[3] ettcp, http://sourceforge.net/projects/ettcp.

[4] KFT, “Using Kernel Function Trace,” http://elinux.org/Using_Kernel_Function_Trace.

[5] M. Rio et al., ”A Map of the Networking Code in Linux Kernel 2.4.20,” Technical Report DataTAG-2004-1. FP5/IST DataTAG Project. 31 March 2004.

[6] TCP/IP call graph, http://zhangyafeikimi.javaeye.com/blog/250511.

[7] Analysis kernel of TCP/IP, http://blog.chinaunix.net/u2/64681/showart_1432551.html.

[8] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman, “Time, Delays, and Deferred Work,” in Linux Device Driver, 3th ed. O'Reilly Media, 2005, pp. 186-188.

[9] Marcus Voelp, “Kernel Entry / Kernel Exit,” http://i30www.ira.uka.de/teaching/coursedocuments/71/Voelp_Kernel-Entry-Kernel-Exit.pps.