Lwip 協定的設計與實現

（中文版）

摘要

LWIP是TCP/IP協定的一種實現。LWIP的主要目的是減少記憶體的使用量和程式碼的大小，使LWIP適合應用於小的、資來源有限的處理器如嵌入式系統。爲了減少處理器和記憶體要求，lwIP可以透過不需任何資料複製的API進行裁減。

本文敘述了lwIP的設計與實現。敘述了協定實現及子系統中所使用的算法和資料結構如記憶體和緩衝管理系統。還封包括LWIP API的參考手冊和使用LWIP 的一些程式碼例子。

目錄

1 序論

在過去的幾年裡，人們對電腦互連和電腦無線互連支援設備的興趣不斷的增長，計算機逐漸與日常使用的設備緊密結合，並且價格不斷下降。同時，無線網路技術如Bluetooth[HNI+98] 和IEEE 802.11b WLAN [BIG+97]不斷顯現。這也在一些領域譬如醫療保健、安全保衛、交通運輸、加工業等引起了許多新引人入勝的情節。小的設備如感應器能被連入現有的網路如全球網際網路，並可以在任何地方對其進行監控。

在過去的幾年裡，互連網技術證明自己具有足夠的靈活性來合併不斷改變的網路的環境。與當初爲低速網路譬如ARPANET網而產生的互連網相比，今天的大範圍連接的互連網技術在頻寬和誤碼率方面都與原來有著巨大的差異。由於互連網的大量應用，把將來的無線互連網路應用於現有的互連網路將會給我們帶來巨大的收益。並且，大面積互連的互連網也是一強勁趨勢。

自從人們經常對像感應器這樣的小設備有小的物理外形和便宜的價格的要求，實現一較少的處理和記憶體要求的互連協定就成爲必須解決的問題。本文描述了一種稱爲LWIP的小到足以滿足最小系統要求的TCP/IP協定的設計與實現。

本文架構如下編排：第2，3和4部分對lwIP堆疊作一個概述，第5部分敘述作業系統類比層，第6部分敘述緩衝器和記憶體管理。第7部分介紹lwIP抽象的網路界面，第8，9，和10部分敘述IP，UDP，和TCP協定的實現。第11和12部分敘述怎樣與lwIP進行界面並介紹lwIP API。第13和14部分分析了實現進程。最後，15部分提供了lwIP API用戶參考手冊，17和18部分展示了多種程式碼例子。

2 協定分層（Protocol layering）

TCP/IP協定被設計爲分層架構，各協定層分別解決通信問題的一部份。這一分層對於協定的設計、實現可起一個指導作用，各個協定可分開實現。然而協定嚴格的按分層架構來實現，各層之間的通訊可能會導致總體性能的降低[[Cla82a] 。爲克服這些問題, 協定的某些內部方面可傳達給其它協定共享，但必須注意，保證只有那些重要訊息才在各層共享。

儘管底層協定或多或少可以進行交叉存取，大部分TCP/IP協定，還是在應用層協定與底層協定之間進行嚴格的區分。在大部分作業系統中，底層協定被作爲與應用層程式具有通訊界面的作業系統核心的一部分。應用程式被看作是TCP/IP協定的抽象，網路通訊與進程間通訊或者文件I/O只有很小的差別。這意味著，因爲應用程式不知道被底層協定所使用的緩衝機制，它不能利用緩衝機制對經常使用的資料進行緩衝。同樣，當應用程式發送資料時，在資料被網路代碼處理前，必須把這些資料從應用程式記憶體區被複製到內部緩衝區。

最小系統中使用的作業系統像lwIP的目標系統在核心和應用進程之間常常並不存在嚴格的保護屏障。這就允許應用程式和底層協定之間使用一種更寬鬆的方案，透過共享記憶體。特別地，應用層可以意識到底層協定所使用的緩衝器處理機制。因此，應用可以更有效地重用緩衝區。而且，既然應用進程和網路程式碼可以使用

相同的記憶體，應用可以直接讀寫內部緩衝器，因此節省了執行複製的開銷。

3 總述（Overview）

正如其他TCP/IP協定的實現，分層協定的設計爲LWIP的設計與實現提供一向導。每一個協議都作爲一個模組來實現，提供一些與其他協定的界面函式。儘管各層分開實現，但正如上面所討論的，爲了同時提升處理速度和記憶體利用兩方面的性能，一些層在設計時違背這一原則。例如：當檢驗一接收到的TCP段（segment）的校驗和（checksum）和分解TCP段時，來源和目的IP位址必須被告知TCP模組。LWIP實現時不是透過函式呼叫把IP位址傳遞給TCP，而是TCP模組透過獲取IP標頭的架構進而自己提取這一訊息。

LWIP有幾個模組組成，除了實現TCP/IP協定的各個模組(IP、ICMP、UDP、和TCP)，同時設計了許多支援模組。這些支援模組組成了作業系統類比層（第5章）、緩衝和記憶體管理子系統（第6章）、網路界面函式（第7章）和一些處理網際網路校驗和的函式。LWIP還封包括關於API的摘要（第12章）。

4 進程模型（Process model）

協定實現的進程模型以把系統劃分成爲不同的進程的方法進行描述。用於實現通訊協定的進程模型使每個協定作爲單一的進程執行。這種模型使用嚴格的協定分層，協定之間的通訊結點必須被嚴格定義。雖然這種方法有其諸多優勢如協定能在執行時被增加，程式碼一般容易理解和調試，但也有不利原素。嚴格的分層，正如先前所述，並不總是實現協定的最好方法。同時，更重要的，每跨越一層，必須做一次上下文切換。這將意味著，接受一個TCP段要進行三次上下文切換：從網路界面的驅動，到IP處理，再到TCP處理，最終到應用處理。根據網路界面的設備驅動程式，對於IP進程，對於TCP進程和最後。在大多數作業系統中一個上下文切換所花的代價都是相當昂貴的。

另一個較普通的方法是把通信協定封裝在作業系統的核心。在這種核心實現通訊協定的情況下，應用程式透過系統呼叫完成通訊。通訊協定之間不嚴格區分，但可以使用交叉協定分層技術。

lwIP所使用的進程模型是：把所以協定封裝到一個單一的進程中，從而與作業系統核心分開。應用程式可能也駐留在lwIP處理進程中，或者在單獨的進程中。TCP/IP堆疊和應用程式之間的通信可以透過函式呼叫實現，也可以透過更爲抽象的API。

以上兩種LWIP的實現方法各有其優缺點。把LWIP作爲一個進程的主要優點是便於在不同的作業系統上移植。由於LWIP的設計目標是面向小的作業系統，這些作業系統一般不支援進程外交換（swapping out processes）或者虛擬記憶體，這樣由於LWIP處理進程交換或者翻頁到磁片而引起的不得不等待磁片回應造成的延遲將不再是一個問題。儘管在獲得服務回應前必須等待調度仍然是一個問題，但是，在LWIP設計時，這並沒有妨礙它在一作業系統核心中實現。

5 作業系統類比層

爲了使lwIP便於移植，與作業系統有關的功能函式呼叫和資料結構沒有在程式碼中直接使用。而是當需要這樣的函式時，作業系統類比層將加以使用。作業系統類比層向諸如定時器、處理同步、消息傳送機制等的作業系統服務提供一套統一的界面。原則上，移植lwIP到其他作業系統時，僅僅需要實現適合該作業系統的作業系統類比層。作業系統類比層提供了由TCP使用的定時器功能。作業系統類比層提供的定時器是一次性的定時器，當超時發生時，呼叫一個已註冊函式至少要200ms的間隔。進程同步機制僅提供了信號量。即使在作業系統底層中信號量不可用，也可以透過其他信號源像條件變量或互鎖來類比。

資訊傳遞的實現使用一種簡單機制，用一種稱爲“郵箱”的抽象方法。郵箱做兩種操作：郵寄和提取。郵寄操作不會阻塞進程；郵寄到郵箱的消息由作業系統類比層排入隊列直到另一個進程來提取它們。即使作業系統底層對郵箱機制不支援，也容易用信號量實現。

6 緩衝和記憶體管理

通訊系統中的記憶體和緩衝管理必須能夠適應大小變化的緩衝區，從幾百位元組的封包含完全大小TCP段的緩衝區到僅僅封包含幾個位元組的短的ICMP回報。而且，爲了避免複製它應當儘可能讓緩衝區的資料內容駐留在記憶體中，網路子系統不管理像應用記憶體或ROM這樣的記憶體。

6.1封包緩衝器－pbufs

Pbuf在lwIP的內部表示一封包，也是爲了最小限度的使用堆疊這一特殊需要而設計。Pbufs類似於用於BSD實現的mbufs。pbuf架構既支援分發動態記憶體來保存封包內容，也支援把封包資料記憶體在靜態記憶體區。Pbufs能在一張列表中一起被連在一起，稱爲一個pbuf鏈，這樣一個封包可以跨越若干個pbufs。

Pbufs具有三種類型，PBUF RAM，PBUF ROM，和PBUF POOL。圖1中pbuf描繪了PBUFRAM類型，和儲存的被pbuf子系統管理的資料封包。圖2中的pbuf是被鏈在一起的pbuf的一個例子，在其中鏈的第一個pbuf具有PBUF RAM類型（where the first pbufin the chain is of the PBUFRAM type），而第二個具有PBUF ROM類型，這意味著它具有不被pbuf系統管理的記憶體資料。

第三種類型的pbuf ，PBUF POOL如圖3所示，封包括從共有的固定大小的pbufs分發的固定大小的pbufs （consists of fixed size pbufs allocated from a pool of fixedsize pbufs.）。一個pbuf鏈可能封包括多重類型的pbufs。

三種類型有不同的使用。PBUF_POOL主要被網路設備驅動程式使用，因爲對作業系統來說分配單一的pbuf速度較快並且適合用於中斷管理（suitable foruse in an interrupt handler）。當應用程式發送位於被應用程式管理的記憶體區的資料時，PBUF_ROM被使用。在pbuf被移交到TCP/IP堆疊後，資料不能修改，因此這一pbuf類型，這類型主要用於資料位於ROM時(因此名稱爲PBUF_ROM)。PBUF_ROM pbuf中的資料可能會用到的頭記憶體在PBUF_RAM pbuf中，它鏈接在PBUF_ROM pbuf的前面，如圖2所示。

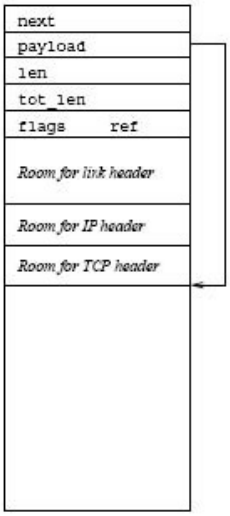


Figure 1. A PBUF_RAM pbuf with data in memory managed by the pbuf subsystem.

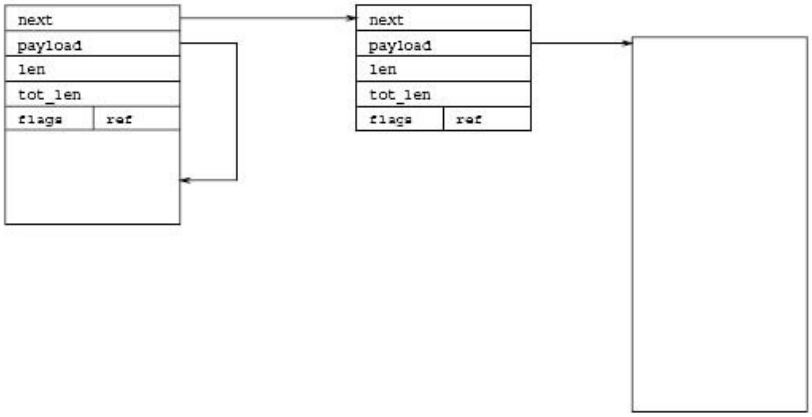


Figure 2. A PBUF_RAM pbuf chained with a PBUF_ROM pbuf that has data in external memory.

當應用程式發送動態地產生的資料時，PBUF_RAM類型的Pbufs也被使用。在這種情況下，pbuf系統不光爲應用資料分發記憶體空間，也爲將要發送的資料的標頭準備空間。如圖1所示。pbuf系統不能預先知道爲將要發送的資料準備什麼樣的標頭，並且假定最壞的情況。標頭大小在編譯時可動態配置。實質上,進入pbufs的是PBUF_POOL類型，離開pbufs的是PBUF_ROM或PBUF_RAM類型。

pbuf的內部的架構如圖1~3。pbuf架構封包括兩個指針，兩長度域，一個flags域，和一參考計數。pbuf鏈中next域是一個指向下一個pbuf的指針。Payload指針指向pbuf中的資料的起始位置。len域封包含pbuf的資料內容的長度。Tot_len域封包含當前的pbuf的長度和在pbuf鏈中接下來的pbufs的所有len領域的總數。換句話說，tot_len域是len域和pbuf鏈中的隨後的pbuf中的tot_len域的值的總和。flags域表明pbuf的類型，而ref領域封包含一參考計數。Next和payload域是內部指針和倚賴於處理器體系架構的資料大小。兩個長度域爲16位無符號整形，flags和ref域均爲4bit寬。pbuf架構整個的大小取決於所使用的處理器體系架構中一個指針的大小及可能的最小alignment的大

小。在帶有32位指針和4個位元組alignment的體系架構，整個的大小爲16位元組，16位指針和1個位元組alignment的體系架構上，大小是9個位元組。

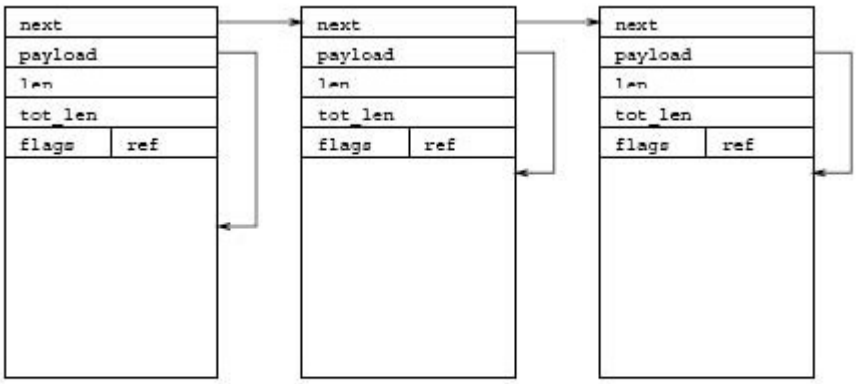


Figure 3. A chained PBUF_POOL pbuf from the pbuf pool.

pbuf模組爲操縱pbufs提供了函式。函式pbuf_alloc() 完成分發一個pbuf的任務，它能夠分發上面所說的三種pbuf中的任何一種。pbuf_ref()增加參考計數。pbuf_free()完成釋放分配的工作，它首先減少pbuf的參考計數。如果參考計數到達零表示pbuf已經被釋放。函式pbuf_realloc()收縮pbuf使它剛好能夠封包含資料大小。pbuf_header()調整payload 指針和長度域，以便對pbuf中的資料的標頭進行預先估計。buf_chain()和pbuf_dechain()用於用鏈接pbufs。

6.2記憶體管理

支援pbuf調度的記憶體管理非常簡單。它處理記憶體中連續區域的分發和釋放，可以緊縮一個預先分發的記憶體塊。記憶體管理器使用系統中總記憶體的專用部分，這確保網路系統不會使用所有可利用記憶體，而且如果網路系統用了所有它自己的記憶體其他程式的操作也不會影響它。

在內部，記憶體管理透過將一種小的架構放置在每一被分發的記憶體塊的頂端上來追蹤分發的記憶體。這個架構（圖4）中設置兩個指針指向記憶體中下一個和前一個分發塊，還有一個used標誌用來指示這個分發塊是否已經被分發。透過搜索一個未使用的記憶體塊來分發記憶體，這個記憶體塊對於請求分發來說足夠大。使用最先適用原則，因此第一塊被使用的記憶體足夠大。當一個分發塊釋放時，used標誌被設爲0。爲了防止碎片，檢測下一個和上一個分發塊的used標誌，如果它們還沒被使用，幾個塊合併成一個大未使用塊。

7網路界面

硬體設備驅動程式中，lwIP用一個類似於BSD的網路界面架構來描述物理硬體。網路界面架構如圖5所示。透過next指針，網路界面被連成一個全局鏈表（global linked list）。每個網路界面有一個名字，記憶體在圖5中的name字段。這個兩個字符的名字識別用於網路界面中的設備驅動類型，而且當界面在執行時由人爲操作來配置。這個名字由設備驅動設置，應當映射由網路界面表示的硬體類型。例如，藍牙

驅動網路界面可能使用名字btd，而IEEE802.11bWLAN硬體可能使用名字wl。由於這些名字不必是唯一的，num字段用來區分同類設備中的不同的網路界面。

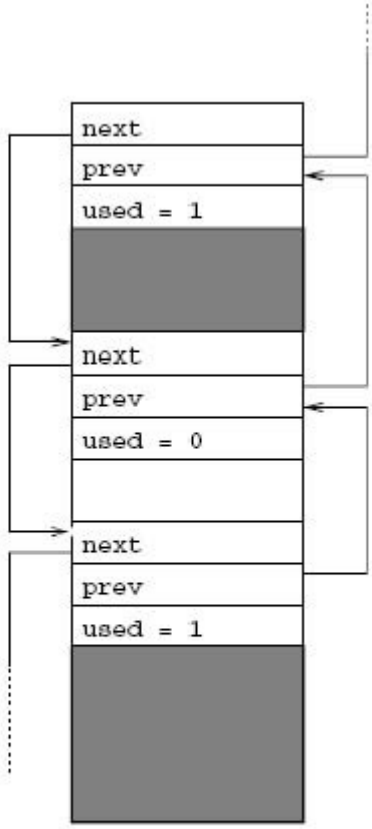


Figure 4. The memory allocation structure.

```
struct netif {
  struct netif *next;
  char name[2];
  int num;
  struct ip_addr ip_addr;
  struct ip_addr netmask;
  struct ip_addr gw;
  void (* input)(struct pbuf *p, struct netif *inp);
  int (* output)(struct netif *netif, struct pbuf *p,
                 struct ip_addr *ipaddr);
  void *state;
};
```

Figure 5. The `netif` structure.

當發送和接收封包時，三個IP位址ip_addr，netmask和gw由IP層使用，它們的用途在下一節敘述。給網路界面配置多於一個IP位址是不允許的，一個網路界面應當爲每一個IP位址創建。當封包接收到時，設備驅動應當呼叫input指針指向的函式。網路界面透過output指針連接到設備驅動。這個指針指向設備驅動中的一個函式，它

在物理網路上傳送一個封包，當一個封包被發送時它由IP層呼叫。這個字段由設備驅動初始化函式填充。

output函式的第三個參數ipaddr是主機的IP位址，它可以接收實際鏈路層的幀。它不應和IP封包的到達站址相同。特別地，當發送一個IP封包到不在本地網路的主機時，鏈路層幀被發送到網路上一個路由。這種情況，給output函式的IP位址將是路由的IP位址。最後，state指針指向設備驅動中網路界面的特定狀態，由設備驅動設置。

8 IP處理

lwIP僅僅實現IP最基本的功能，它可以發送，接收和轉發封包，但不能發送或接收分割的IP封包，也不能處理帶IP選項的封包。對於大多數應用來說這不會引起任何問題。

8.1接收封包

對於接收的IP若干封包，處理從ip_input( )函式被設備驅動程式呼叫開始。在這裡，初始化工作將檢查IP版本，同時確定標頭長度，還會計算和檢查標頭checksum域。期望的情況是，自從Proxy伺服器重組所有碎片（fragmented）封包以來，堆堆疊就再沒有收到碎片(fragments)，這樣任何IP碎片的封包都會被默默的丟棄。帶有ip選項的封包同樣會被指定爲由代理處理，並因此被丟掉。

接下來，函式透過網路界面的IP位址檢驗到達站址以確定封包是否去往主機。網路界面已在鏈表中排序，可以線性查找。網路界面的序號指定爲是小的號，因爲比線性查找更巧妙的查找方法還沒實現。如果接收的封包是主機指定的封包，將使用protocol域來決定該封包應該傳給哪個更高層協定。

8.2發送封包

一個要發送的封包由函式ip_output( )處理，它使用函式ip_route( )尋找適當的網路界面來上傳封包。當時發送封包的網路界面被確定後，封包被傳遞到ip_output_if()函式，該函式把發送網路界面作爲一個函式自變量。在這裡，所有IP標頭域被填補並且IP標頭checksum被計算。IP封包的來源和到達站址作爲變量傳遞給ip_output_if()函式。來源位址可能被略去（left out），然而，在這種情況下要發送的網路界面的IP位址將被用作封包的來來源IP位址。

ip_route()函式透過線性查找網路界面列表找到適合的網路界面。在查找IP封包的目的IP地址期間，用網路界面的網路遮罩進行遮罩。如果到達站址等於經遮罩的界面IP位址，則選擇這個界面。如果找不到匹配的，則使用預設網路界面。預設網路界面由人工操作在啓動時或執行時配置。如果預設界面的網路位址和目的IP位址不匹配，則選擇網路界面架構中的gw字段作爲鏈路層幀的目的IP位址。（注意這各情況下IP封包的到達站址和鏈路層幀的IP位址是不同的。）

路由的原始形式忽略了這個事實：一個網路可能有許多路由器依附它。而工作時，對於一般情況下，一個本地網路只有一個路由器。因爲運輸層協定UDP和TCP在計算運輸層校驗和時需要有目的IP位址，所以在封包傳給IP層前外發網路界面在

某些情況下必須已確定。這可讓運輸層函式直接呼叫ip_route()函式完成，因爲在封包到達IP層時外發網路界面已經知道，沒必要再查找網路界面列表。而是那些協定直接呼叫ip_output_if()函式。由於這個函式把網路界面作爲參數，可避免外發界面的查找。注：執行期間lwIP的手工配置要有一個能配置堆疊的應用程式，lwIP中不封包含這樣的程式。

8.3轉交封包

如果沒有網路界面的IP位址和傳進封包的到達站址相同，這個封包應當轉發。這由函式ip_forward()完成。在這裡，TTL字段減小，如果變爲零，則ICMP錯誤訊息被發送到IP封包原始發送器並丟棄這個封包。由於IP頭被改變，有必要調整IP頭校驗和。然而不必重算完整的校驗和，因爲可用簡單的算術來調整原始的IP校驗和[MK90,Rij94]。最後，封包被轉發到適當的網路界面。用來尋找合適網路界面的算法和發送IP封包時使用的一樣。

8.4 ICMP處理

ICMP處理是相當簡單的。由ip_input()收到的ICMP封包被移交到icmp_input()，它解析ICMP標頭並且進行適當的處理。一些ICMP訊息被傳遞到更高協定層並被傳輸層的一些特殊函式處理（Some ICMP messages are passed to upper layer protocols and those are taken care ofby special functions in the transport layer）。ICMP到達站不能到達的訊息能被運輸層協議發送，尤其是UDP，和函式icmp_dest_unreach( )。

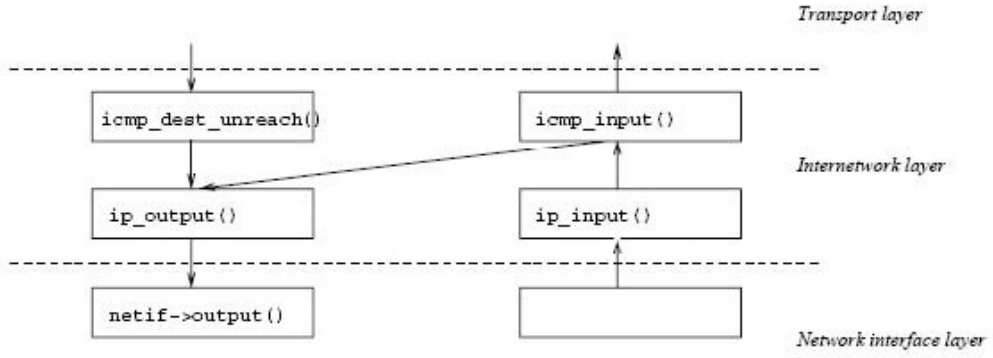


Figure 6. ICMP processing

使用ICMP ECHO訊息來探測網路被廣泛應用，因此對ICMPECHO進行了性能優化。實際處理在函式icmp_input( )中發生，封包括對接收的封包的目的和來源位址進行交換，改變ICMP類型爲echoreply並且調整ICMP checksum。然後封包回送到IP層等待傳送。

9 UDP 處理

UDP 是一個簡單的協定，用來完成不同處理進程間的封包分離。每一個UDP話路(session)的狀態都被保留在一個PCB 架構中，如圖7所示。UDP PCBs 保存在一個鏈表中，當UDP datagram到達，則搜索該鏈表並進行匹配。

UDP PCB 架構中封包含一個指向全局UDP PCB鏈表中的下一個PCB的指針。UDP話路(session)由IP位址和端口號來定義，並且被存放在local_ip, dest_ip,local_port,dest_port域中。Flags域指出這一話路（session）將使用什麼樣的UDP校驗和策略。這可能既沒關掉UDP checksumming 完全，或者使用UDP 輕便在哪個檢驗數字蓋住只資料報的部分。This can be either to switch UDP checksumming off completely, or to use UDP Lite [LDP99] in which the checksum covers only parts of the datagram. If UDP Lite is used, the chksum lenfield specifies how much of the datagram that should be checksummed.當接收到由PCB標明的session中的datagram時，最後二個參數recv 和recv_arg將被使用。當接收到datagram時，recv所指向的函式被呼叫。

```
struct udp_pcb {
  struct udp_pcb *next;
  struct ip_addr local_ip, dest_ip;
  u16_t local_port, dest_port;
  u8_t flags;
  u16_t chksum_len;
  void (* recv)(void *arg, struct udp_pcb *pcb, struct pbuf *p);
  void *recv_arg;
};
```

Figure 7. The udp_pcb structure

由於UDP較爲簡單，輸入和輸出處理也較簡單，如圖8所示。

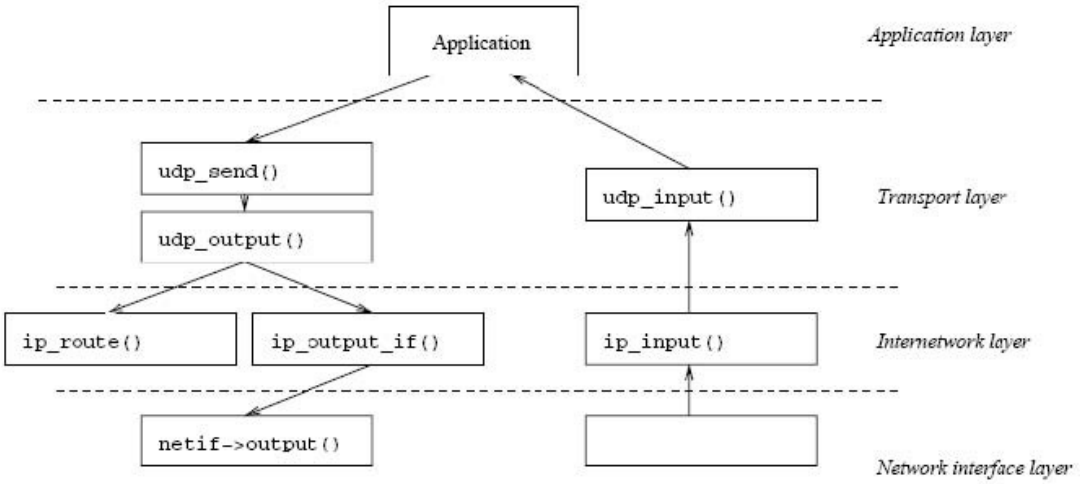


Figure 8. UDP processing

當收到一個UDP datagram 時，IP層呼叫udp_input()函式。這裡，如果session應該使用checksumming，UDP checksum將被檢查同時datagram被分離。當發現相應的UDP PCB，recv函式被呼叫。

10 TCP 處理

TCP 爲傳輸層協定它爲應用層提供可靠的二進製資料流服務。TCP協定比這裡描述的其它協定都要複雜, 並且TCP 程式碼占lwIP總程式碼的50%。

10.1 總述

基本TCP 處理(圖9) 被劃分成六個函式；函式tcp_input()、tcp_process()、tcp_receive()與TCP 輸入處理有關, tcp_write()、tcp_enqueue()、tcp_output() 對輸出進行處理

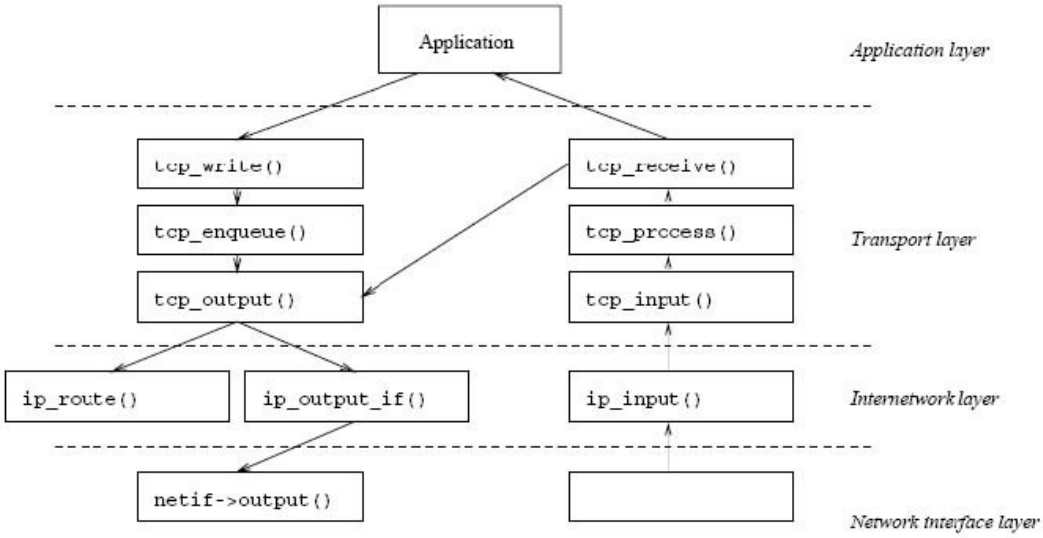


Figure 9. TCP processing

當應用程式想要發送TCP 資料, 函式tcp_write()將被呼叫。函式tcp_write() 將控制權交給tcp_enqueue()，該函式將資料分成合適大小的TCP段（如果必要），並放進發送隊列。接下來函式tcp_output()將檢查資料是否可以發送。也就是說，如果接收器的窗口有足夠的空間並且擁塞窗口足夠大,則使用ip_route()和ip_output_if() 兩個函式發送資料。當ip_input()對IP標頭進行檢驗且把TCP段移交給tcp_input()函式後，輸入處理開始。在該函式中將進行初始檢驗(也就是,checksumming 和TCP 剖析析)並決定該段屬於哪個TCP連接。

該段於是由tcp_process()處理,它實現TCP狀態機和其他任何必須的狀態轉換。。如果一個連接處於從網路接收資料的狀態，函式tcp_receive() 將被呼叫。如果那樣, tcp_receive() 將把段上傳給應用程式。如果段構成未應答資料（先前放入緩衝區的）的ACK, 資料將從緩衝被移走並且收回該記憶體區。同樣, 如果接收到請求資料的ACK，接收者可能希望接收更多的資料，這時tcp_output() 將被呼叫。

10.2 資料結構

由於小型嵌入式系統記憶體的限制，LWIP所使用的資料結構被故意縮小。這是在資料結構複雜度與使用資料結構的程式碼的複雜度之間的一個折衷。這樣就因爲要保證資料結構的小巧而使程式碼複雜性增加。

TCP PCB相當大，如圖10 。因爲TCP 連接在處於監聽（listen）和時間等待（TIME-WAIT）狀態時比處於其他狀態的連接需要保留較少狀態訊息，對於這些連接使用了一種更小的PCB 數據架構。這種資料結構鑲嵌在完整的PCB 架構中, 在PCB 架構中的排列持續如圖10， 因此有些笨拙。

TCP PCBs 被保留在一份鏈表中, 並且next指針把PCB列表連接在一起。狀態變

量封包含當前連接的TCP狀態。其次, 辨認連接的IP 位址和端口號被保存。mss 變量保存連接所允許的最大段大小。

當接受資料時，rcv_nxt 和rcv_wnd域被使用。rcv_nxt域封包含期望從遙端的下個順序編號（contains the next sequence number expected from the remote end），因而當發送ACKs 到遠程主機時被使用。接收器的窗口被保留在rcv_wnd中，並且在將要發出的TCP 段中被告知。tmr被作爲定時器使用，在經過一特定時間後連接應該被取消, 譬如連接在TIME-WAIT 狀態。連接所允許的最大段大小被存放在mss域中。Flags域封包含連接的附加狀態訊息, 譬如連接是否爲快速恢復或被延遲的ACK是否被發送。

```
struct tcp_pcb {
  struct tcp_pcb *next;
  enum tcp_state state;     /* TCP state */
  void (* accept)(void *arg, struct tcp_pcb *newpcb);
  void *accept_arg;
  struct ip_addr local_ip;
  u16_t local_port;
  struct ip_addr dest_ip;
  u16_t dest_port;
  u32_t rcv_nxt, rcv_wnd;   /* receiver variables */
  u16_t tmr;
  u32_t mss;                /* maximum segment size */
  u8_t flags;
  u16_t rttest;             /* rtt estimation */
  u32_t rtseq;              /* sequence no for rtt estimation */
  s32_t sa, sv;             /* rtt average and variance */
  u32_t rto;                /* retransmission time-out */
  u32_t lastack;            /* last ACK received */
  u8_t dupacks;             /* number of duplicate ACKs */
  u32_t cwnd, u32_t ssthresh;  /* congestion control variables */
  u32_t snd_ack, snd_nxt,   /* sender variables */
    snd_wnd, snd_wl1, snd_wl2, snd_lbb;
  void (* recv)(void *arg, struct tcp_pcb *pcb, struct pbuf *p);
  void *recv_arg;
  struct tcp_seg *unsent, *unacked, /* queues */
        *ooseq;
};
```

Figure 10. The tcp_pcb structure

域rttest, rtseq, sa, 和sv 被使用爲round-trip時間估計。用於估計的段順序號記憶體在rtseq ，該段被發送的時間存放在rttest。平均round-trip時間和round-trip時間變化分別存放在sa和sv 。這些變量被用來計算存放在rto域中的轉播暫停（retransmission time-out）。

二個域lastack 和dupacks 被用來實現快速轉播和快速重發。lastack封包含由最後接受到的ACK應答的順序編號，dupacks 對接收到多少關於記憶體在lastack的順序編號的ACK 進行計數。當前連接的阻塞窗口被存放在cwnd域，並且緩慢的起動門限被保留在ssthresh 。

六個域snd_ack、snd_nxt、snd_wnd、snd_wl1、snd_wl2 和snd_lbb 在發送資料時使用。由接收器應答的最高的順序編號被存放在snd_ack， 並且下個被發送的順序

編號保存在snd_nxt 。接收器的廣告窗口(advertised window)保存在snd_wnd，兩域snd_wl1 和snd_wl2在更新snd_wn時使用。Snd_lbb域封包含發送隊列最後位元組的順序編號。當傳遞接受的資料到應用層時將使用函式指針recv 和recv_arg。三個隊列unsent, unacked和looseq當發送和接受資料時被使用。已經從應用接收但未被發送的資料由隊列unsent進行排隊，已發送但還沒有被遠程主機應答（acknowledged）的資料由unacked 記憶體。接收的序列外面的資料由looseq進行緩衝。

```
struct tcp_seg {
  struct tcp_seg *next;
  u16_t len;
  struct pbuf *p;
  struct tcp_hdr *tcphdr;
  void *data;
  u16_t rtime;
};
```

Figure 11. The tcp_seg structure

表11中的Tcp_seg 架構是TCP 段的內部表示方法。這個架構由一個next指針開始，該指針用於連接排隊段。len域封包含段的長度。這意味著一個資料段的len域將封包含段中資料的長度, 具有SYN或FIN標誌的空段的len被設置爲1 。pbuf架構類型指針p 是封包含實際段、tcphdr、指向TCP頭的資料指針和資料段的緩衝。分別的，對於將要外發的段, rtime域被用於該段的轉播暫停。因爲接收的段不會需要被轉播, 對於接收段來說這個域並不需要並且也不爲該域分發記憶體。

10.3 順序編好計算（Sequence number calculations）

用於枚舉TCP二進製資料流的TCP順序編號爲無符號32位資料， 因此其範圍是$[0, 2^{32} - 1]$。因爲在TCP 連接中要發送位元組的數量可能會比32 位組合的數量更多, 順序編號以2^32爲模數進行計算。這意味著, 普通的比較操作符無法被TCP順序編號使用。修改過的比較操作符, 叫做<seq和>seq, 由下面關係定義：

$$s <_{seq} t \Leftrightarrow s - t < 0$$

$$s >_{seq} t \Leftrightarrow s - t > 0,$$

這裡s 和t 是TCP 順序編號。比較操作符≦≧也等效地被定義。比較運算子在C全域定義在標頭文件。

10.4 隊列和發送資料

將要發送的資料被劃分成適當的大小的大塊並tcp_enqueue()指定順序編號。這裡, 資料被打封包進pbufs 架構並附加進tcp_seg 架構。在pbuf內，TCP頭被建立，並且除應答數字, ackno和廣播窗口, wnd以外的所有域被填充。這些域在段排隊時可以被改變，並由tcp_output()進行設置，該函式完成段的實際傳輸。在段被建立之後, 他們被送往PCB裡的unsent列表進行排隊。函式tcp_enqueue()設法用最大段大小的資料填裝各段直到在unsent 隊列的末端發現段under-full, 該段使用pbuf chaining functionality功

能被添附以新資料。

在tcp_enqueue() 格式化和排隊了段之後, tcp_output()函式被呼叫。它檢查在當前的窗口中是否還有空間來記憶體更多的資料。當前窗口的數值是透過窗口資料擁擠的最大量和接收窗口的廣播。(It checks if there is any room in the current window for any more data. The current window is computed by taking the maximum of the congestion window and the advertised receiver's window) 。其次, 它填裝由tcp_enqueue() 未填裝的TCP頭的域，並且使用ip_route()和ip_output_if()傳送段。在段被放入傳輸後unacked表後,停留直到該段的ACK被接收到。當段放入unacked 表的同時, 如在10.8部分所描述同時也爲重發計時。當段需重發時原來的TCP、IP頭被保留，只需對TCP 頭做少量變化。TCP頭的ackno 和wnd域被設置爲當前值，因爲在段的原始傳輸和重發期間我們可能接收了資料。這只改變標頭裡的二個16 位字和整體TCP checksum不必要重新計算，因爲簡單的算術[Rij94] 可以用來更新checksum。

當段最初被傳送時，IP 層已經增加了IP 頭，並且沒有理由改變它。因而重發不要求IP頭的checksum.的任何重發計算。Silly Window Syndrome [Cla82b] (SWS)綜合症狀[Cla82b] (SWS) 是一個可能導致非常壞的性能的TCP 現象。當TCP 接收器廣播一個小窗口並且TCP 發送者立刻發送資料填裝窗口時SWS發生。當這小段被應答窗口再次打開並且發送者將再發送小段填裝窗口。這導致TCP 資料流封包括許多非常小段的情況發生。爲了避免SWS 發送者和接收者都必須設法避免這個情況。接收者不能給小窗口更新做廣播並且當只提供一個小窗口時發送者不能送小段。

在lwIP， SWS 在發送端自然地被避免，因爲TCP 段被建立和排隊時沒有做廣播接收器窗口的應答。在大資料量傳輸時隊列將封包括最大尺寸大小的段。這意味著如果TCP 接收器廣播一個小窗口, 發送者不會送隊列中的第一個段，因爲它比廣播的窗口大。相反, 它將等待直到窗口是足夠大以至於能容納最大大小的段。當作爲TCP 接收器時, lwIP 不會給比連接的最大段大小小的接收器的窗口做廣播。

10.5 接收段

10.5.1 複用

當TCP 段到達tcp_input() 函式時, 他們將在TCP PCBs之間被解析（demultiplexed） 。解析的關鍵是來來源和到達站IP 位址和TCP 端口數。當解析段的時候有種二類型的PCBs必須突出的（distinguished）：那些對應於開放連接的和那些對應於連接是半的打開。半開放連接是指那些處於監聽狀態和只有本地TCP端口號被指定且本地IP 位址爲任意值的連接, 但是開放連接有指定的兩個IP 位址和兩個端口號。

許多TCP 的實現, 譬如早期的BSD 實現, 使用具有單一入口緩衝器的PCBs鏈表技術。在這之後基本理論是, 多數的TCP 連接構成批量傳送，它典型地顯示一個大批量的位置[Mog92], 造成一個高緩衝命中比率。另一個方案封包括兩個具有單一入口的緩衝器, 一個爲對應於被送的最後的封包的PCB，一個爲最後接受的封包的PCB [PP93] 。透過移動最近被使用的PCB 到列表的前端，一份供選擇的方案可以實施。兩種方法[MD92]都勝過單一入口的緩衝方案。

在lwIP, 每當PCB 匹配被發現，解析段時, PCB都將被移動向PCBs 列表的前

端。但是，在聽狀態的連接所用的PCBs並不被移動, 因爲這種連接並不期望接收段。

10.5.2 接收資料

對接收到的段的實際處理是在函式tcp_receive()裡進行的。段的應答數字與在unacked隊列中的段比較。如果應答數字比段在unacked 隊列的順序編號高, 該段從隊列中被移走並且爲段分發的記憶體也被收回。

如果接收的段的順序編號比PCB中rcv_nxt變量高，該段將脫離序列。序列外的段在PCB中的ooseq 隊列進行排隊。如果接收的段的順序編號與rcv_nxt 是相等的, 透過呼叫在PCB中的函數recv，段被轉交給上層，並且關於接收段的長度的rcv_nxt域被加進來。因爲在序列內的段的接收也許意味著先前被接受的在序列外的段是被期望的下一個段, ooseq 隊列被檢查。如果它封包含順序編號與rcv_nxt順序編號相等的段, 透過呼叫函式recv該段被轉交給應用程式並且更新rcv_nxt。這個進程持續到ooseq 隊列爲空或ooseq的下一個段脫離序列。

10.6 接受新的連接（Accepting new connections）

處於聽狀態，也就是說已經被動地開放的TCP 連接，, 已經準備從遠程主機接受新的連接。爲了那些連接，必須建立新的TCP PCB ，並傳遞給打開初始聽連接的應用程式。在lwIP裡， 這個步驟是透過使用回調函式來完成，這個函式當一個新的連接被建立時被呼叫。

當處於聽狀態的連接接收一個具有SYN標誌的TCP段時（When a connection in the LISTEN state receives a TCP segment with the SYN flag set），一個新的連接就建立了，並且一個帶有SYN和ACK的標誌的段被送出以回應那個SYN段。這個時候這個連接就進入了SYN-RCVD狀態，並等待發送的SYN段的應答，當應答訊息收到後，這個連接就進入了ESTABLISHED階段，接受函式（在PCB中的accept域如圖10）會被呼叫。

10.7 快速轉發（Fast retransmit）

IWIP裡擁有快速轉發和快速恢復的功能。該功能透過明確最後被應答的順序編號實現的。如果收到同一順序編號的另外個應答信號，TCP PCB 裡的dupacks 計數建立。當DUPACKS到3的時候，在未應答序列中的第一個段將要被重新送出，快速恢復被初始化。快速恢復的步驟完成以後的動作[ASP99]：無論什麼時候收到一個新資料的應答信號，dupacks計數都複位爲0。

10.8 定時器（Timers）

和在BSD TCP 中實現一樣，lwIP 使用二個週期性定時器，週期分別爲500ms和200ms。這二個定時器又被用來實現更加複雜的邏輯定時器，如轉發定時器、TIME-WAIT 定時器和delayed ACK定時器。

fine grained timer定時器，如果有應該被發送的任一被延遲的ACKs，tcp_timer_fine()將在定時結束時檢查TCP PCB，正如在tcp pcb 架構中的flag域所表明的(圖10) 。如果delayedACK 標誌被設置, 空的TCP 應答段被發送並且標誌被清除。

coarse grained timer定時器，在tcp_timer_coarse()裡實現，並且掃描PCB 列表。對

於任何一個PCB, 未應答段列表(在tcp_seg 架構中的unacked指針，如表11), 被否認(is traversed）,並且rtime 變量被增加。如果rtime 比當前的轉發暫停時間（由PCB架構中的rto 變量給出）大， 段被轉發並且轉發暫停被加倍。只有在擁塞窗口和廣播接收器的窗口的值允許的情況下段才被轉發。在轉發以後, 擁塞窗口被設置成一最大段大小，slow start threshold被設置爲有效的窗口大小的一半，並且slow start在連接中被初始化。

對於在TIME-WAIT狀態的連接, coarse grained timer定時器也會在PCB 架構中增加tmr域。當這個定時器到達2×MSL門限，連接被取消。coarse grained timer定時器同時也增加一個全局TCP 時鐘, tcp_ticks。這個時鐘用來估計round-trip時間轉播暫停（time－out）。

10.9 Round-trip 時間估計（Round-trip time estimation）

round-trip時間估計是TCP 的一個重要部份，因爲估計的round-trip時間被用來確定適當的轉發暫停。在lwIP， round-trip時間測量的實現與BSD 相似。每一次round-trip往返round-trip時間就被測量一次，並且使用smoothing函式（在[Jac88]中描述） 對適當的轉發暫停進行計算。

TCP PCB的變量rtseq 保存著往返時間被測量的段的順序編號。PCB中的Rttest變量保存著當段第一次被傳送時tcp_ticks的值。當一個順序編號等於或大於rtseq的ACK被接收到時，round-trip時間透過從tcp_ticks減去rttest被測量。如果轉發發生在round-trip時間測量期間，測量不被採取。

10.10擁塞控制（Congestion control）

擁塞控制的實現是出乎意料的簡單，僅僅封包括幾行程式碼。當一個新資料的ACK被擁塞窗口接受,cwnd, 被mss2/cwnd 增加或被一最大段大小增加, 取決於連接是緩慢起動還是擁塞避免（depending on whether the connection is in slow start or congestion avoidance）。當發送資料時，接收器的廣播窗口和擁塞窗口的最小值用來確定每個窗口能夠發送資料的多少。

11堆疊界面（Interfacing the stack）

使用由TCP/IP協定堆疊提供的服務有二種模式；一種是直接呼叫在TCP 和UDP模組中的函式,另一種就是使用lwIP API。

TCP 和UDP 模組提供一個網路服務的基本界面。該界面基於回調，因此使用它的應用程式可能因此不必以連續模式進行操作。這使應用程式的編程更加困難並且應用程式碼更難理解。爲了接受資料，應用程式登記一個協定堆疊的回調函式。回調函式同一個特定的連接連繫在一起，當該連接的封包到達時, 回調函式被協定堆疊呼叫。

此外, 與TCP 和UDP 模組直接界面地應用程式，必須(至少部份地)保留在像TCP/IP協定堆疊這樣的處理進程中。這歸結於回調函式無法橫跨處理界限呼叫的事實。這既有好處也有不足。好處是應用程式和TCP/IP 協定是在同一個處理進程中, 發送和接收封包時不用上下文切換。主要不足是，在任何長的連續計算進程中應用程式無法介入自己，因爲TCP/IP 處理無法與計算平行發生， 因而喪失通訊性能。

透過把應用程式分成兩部分可以克服這一缺點, 一部分應付通信一部分應付計算。負責通訊的部分駐留在TCP/IP 進程中，負責計算的部份將是一個單獨的過程。將要在下一節介紹的lwIP API 提供了一個架構化的模式，用這樣模式來劃分應用程式。TCP/IP的處理不應該與其他運算並行處理,這樣將會降低通訊的性能，所以我們把應用程式分解爲兩個部分,一部分專注於處理通訊,另一部分做其他的運算。通訊的部分將封包含在TCP/IP進層中，而其他的繁雜運算則作爲一個獨立的進程。下一節將介紹LWIP 的API 提供的分開的架構來應用的辦法。

12 應用程式界面API

作爲高級別的BSD socket API,它是不適宜用於一個最小限度的TCP/IP執行的.特別BSD SOCKETS要求在TCP/IP協定堆疊中將要發送的資料從應用程式複製到內部緩衝器.需要複製資料的原因是通常TCP/IP協定堆疊和應用程式一般都處在不同的保護領域.大多數時候應用程式是位於戶進程而TCP/IP卻在作業系統核心中。透過避免這額外的複製就可以大幅度的提升API的性能[ABM95]。同樣地,這樣的複製需要分發額外的記憶體，每個訊息封包都浪費了雙倍的記憶體。

LWIP API是專爲LWIP設計並利用LWIP的內部架構達成效果，LWIP API 和BSD API非常類似，但操作相對低級。LWIP API不需要TCP/IP和應用程式之間的相互複製資料，應用程式可以巧妙的直接處理內部緩衝器。

由於BSD SOCKET API 很容易理解且已經有很多人爲它寫過應用程式，LWIP API很有必要有與BSD SOCKET 的兼容層面。17節仲介紹了如何用LWIP API 去重寫BSD SOCKET 函式，15節中有LWIP API 的一個參考手冊。

12.1 基本概念

從應用程式去看，BSD SOCKET API的資料處理都是在一個連續的記憶體區域完成的。這是因爲應用程式通常也是在這樣的一個連續大記憶體塊中完成資料處理的。LWIP採用這樣的機制是沒有優勢的，因爲IWIP通常處理的資料緩衝器都被分割成了小的記憶體塊。在透過應用程式前這些數據就會不得不要複製到一個連續的記憶體區，這將浪費雙方的處理時間和記憶體,因此LWIP API允許應用程式巧妙地直接處理分離的緩衝器去避免額外的複製。

LWIP API 儘管非常類似BSD SOCKET API，可是卻有著值得注意的不同的地方，應用程式使用BSD SOCKET API 時候不需要知道普通文件和網路連接之間的差別，但使用LWIP API的時候就必須要知道確實在使用網路連接。

網路資料被接收到分離的記憶體塊的時候是以緩衝器的形式出現的，由於很多應用程式都希望在一個連續的記憶體區域處理資料，這就要有個函式去把這些緩衝器碎片複製到連續的記憶體空間中。

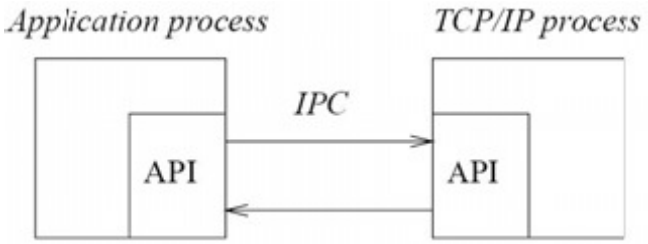
發送出去的網路資料根據是TCP連接還是UDP是不同地處理的。在TCP連接時，資料透過一個指向連續記憶體區的指針發送，TCP/IP協定爲傳送區分適當大小的資料封包和資料隊列。在發送UDP資料的時候，應用程式將明確地分發緩衝器並填充資料，在輸出函式被呼叫的時候由TCP/IP協定堆疊馬上發送出去。

12.2 API的執行

API是分做兩個部分執行的,在圖12中我們可以看到，一部分位於在TCP/IP的進程模組中,一部分當作連接庫在應用程式中執行，這兩部分的API透過由作業系統仿真層提供的進程間通訊(IPC)傳達訊息.當前使用的IPC機制有以下3種：

- ▣ 共享記憶體 (shared memory)
- ▣ 消息 (message passing)
- ▣ 信號 (semaphore)

當作業系統支援這些IPC類型的時候,並不意味著他它們得到了作業系統的最底層的支援,因爲作業系統並不是一開始就支援它們,只是作業系統的的仿真層仿真了它們。



圖示12 . 分開爲兩部分的API執行

一般的設計原理都是儘可能地提升TCP/IP進程裡的API的工作能力更勝於應用程式裡API.這很重要，因爲大部分的進程都用TCP/IP進程進行它們的TCP/IP通訊。遵循API部分的程式碼足跡，和應用程式連接的這部分並不是很重要.這些程式碼可以在進程間共享,即使作業系統並不支援共享連接庫。這些程式碼還可以保存在ROM中，嵌入系統一般儘管處理能力不高卻擁有很大的ROM空間。

緩衝器管理器位於API執行庫裡，buffer在應用程式進程中被建立,複製和分發.應用程式和TCP/IP進程間使用共享記憶體來傳遞buffer,應用程式中用於通訊的buffer資料類型是一種提取於pbuf的類型。buffer傳輸引用到的記憶體，和分發的記憶體不同,它是可以利用共享記憶體的.所以可以進程間共享引用的記憶體.執行LWIP的嵌入式作業系統一般有意不做任何形式的記憶體保護，所以不會有問題。

操作網路連接的函式位於TCP/IP進程的API中.應用程式的API函式會透過一個簡單的通訊協定傳遞一個訊息個TCP/IP進程的API，這訊息封包含操作的類型,和操作的相關變量.這個操作由TCP/IP的API傳輸後用訊息給應用程式一個返回值。

13 程式碼統計分析

本節分析了LWIP的來源程式碼行的數量和編譯後結果大小的關係.程式碼被兩種不同的處理器結構編譯：

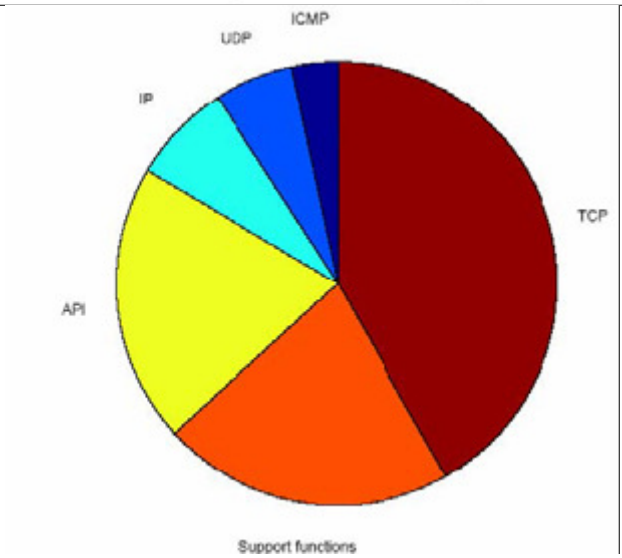
Intel Pentium III處理器,在FreeBSD 4.1下面用GCC 2.95.2 編譯,開啓了編譯優化選項。6520處理器[Nab, Zak83].用cc65 2.5.5 [vB] 編譯,開啓了編譯優化選項。Intel x86 用了7個32位暫存器和32位指針。6502主要用於嵌入系統,具有一個8位累積器兩個8位索引暫存器和16位指針。

13.1 程式碼行

表1

Table 1. Lines of code.

Module	Lines of code	Relative size
TCP	1076	42%
Support functions	554	21%
API	523	20%
IP	189	7%
UDP	149	6%
ICMP	87	3%
Total	2578	100%



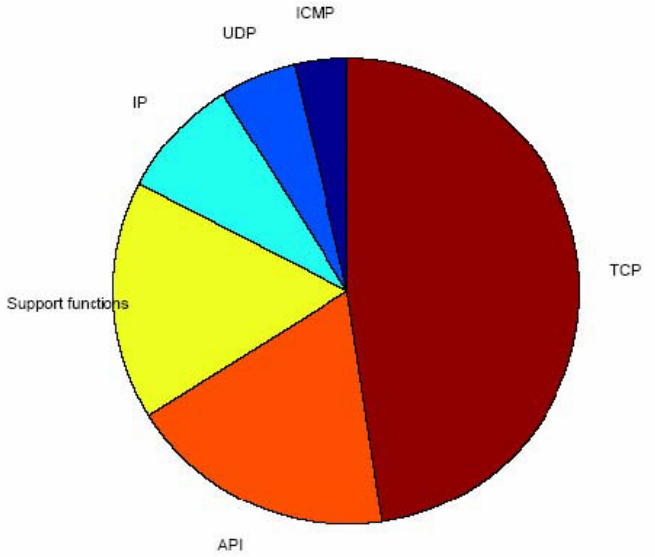
圖示13 程式碼行比例

表1介紹了LWIP程式碼行的數量圖示13顯示了各部分的程式碼比例。"sunpport functions"類別封包括了緩衝器和記憶體管理器還有校驗和處理函式，在事實配置支援的情況下，校驗和函式應該用普通C執行而不用處理器特殊的運算法則。"API"類別封包括了應用程式的API和TCP/IP協定堆疊的API.作業系統的仿真層在這裡沒有分析顯示，因爲它在作業系統的底層而且大小有很大不定性,在這裡就沒必要比較它了。作爲比較，這裡忽略了所有的註釋，空白行和頭文件.我們可以看到TCP這一塊比其它的執行協定都大得多，,而API和support functions 加起來就和TCP差不多大。

13.2 Object Code Size

表2. LWIP在Intel x86下編譯後程式碼的大小

Module	Size (bytes)	Relative size
TCP	6584	48%
API	2556	18%
Support functions	2281	16%
IP	1173	8%
UDP	731	5%
ICMP	505	4%
Total	13830	100%



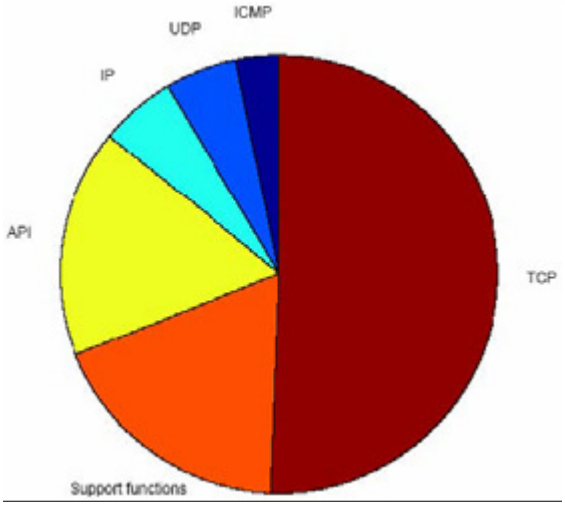
圖示14. LWIP在X86下編譯後程式碼的大小。

表2總結了在Intel x86下編譯後程式碼的大小，圖示14 顯示了相對大小比率.我們可以看到項目的排序有了點小變化。這裡API大於support functions 類別，儘管support functions的程式碼也增加了很多。我們也可以看到TCP的編譯後大小占了總大小的48%但來源程式碼行卻只占總來源程式碼行數的42%。

經檢查TCP模組的彙編輸出發現了一種可能性，TCP模組裡封包括了大量的解除參照指針(原文pointer dereferencing)在很多彙編程式碼行裡，因此增加了編譯後程式碼的大小。很多這些解除參照指針被每個函式用了2或3次。透過修改來源程式碼讓這些指針在一個地方只解除參照(dereferenced)一次然後放進一個本地變量裡就應該可以達到優化目標程式碼的目的。當減少了編譯後目標程式碼的大小後,可能要爲堆堆疊分發更多的記憶體來存放這些本地變量。

表3. LWIP 在6502下編譯後的目標程式碼的大小。

Module	Size (bytes)	Relative size
TCP	11461	51%
Support functions	4149	18%
API	3847	17%
IP	1264	6%
UDP	1211	5%
ICMP	714	3%
Total	22646	100%



圖示15. LWIP 在6502下編譯後目標程式碼的大小比例

表3 列出了在6502下編譯後目標程式碼的大小。圖示14 顯示了各部分的大小比率。我們可以看到TCP，API和support functions幾乎是在Intel x86下編譯的兩倍，然爾IP，UDP和ICMP卻幾乎一樣的大小。我們也可以看到和表2相反過來support functions比API還大,API和support functions的程式碼大小的差別變小了。

TCP模組大小增加的原因主要是6502本來並不支援32位的整數，因此編譯器把每個32位的操作展開爲多行彙編程式碼。TCP序列號就是一個32位的整數而TCP模組執行了多次使用序列號的計算。

我們可以將LWIP裡面TCP的程式碼大小和其他的TCP/IP堆疊的TCP程式碼進行比較，比如和現下很流行的Free BSD4.1下的BSD TCP/IP和Linux2.2.10下的TCP相比較。在Intel x86下用打開優化選項用gcc編譯，LWIP的TCP的執行程式碼接近6600bytes,FreeBSD 4.1的TCP的目標執行程式碼大概27000bytes,幾乎是LWIP的4倍。在Linux 2.1.10下，TCP的編譯後目標執行程式碼高達39000bytes,大概是LWIP的6倍。這樣大的差別事實上是因爲FreeBSD和Linux2.2.10封包含了很多TCP的特性象SACK[MMFR96]與BSD的socket API.在這裡我們不再做IP的執行程式碼的大小比較，因爲在FreeBSD和linux中封包含了大量的IP執行特性，比如：FreeBSD和linux支援防火牆和IP執行隧道，還有動態路由表等，這些都是LWIP中沒有的。

LWIP的API的程式碼量大概是LWIP總程式碼量的六份之一，因爲LWIP可以在沒有API的情況下運行，系統配置的時候可以略掉這部分而只佔用很小的程式碼空間。

14 性能分析

LWIP的記憶體使用和程式碼效率性能沒有在本文中做非常正式的測試，這將在以後大家使用中體會，我們做了個簡單測試，我們用LWIP執行一個簡單的HTTP/1.0的WEB伺服器，在少於4K的RAM下可以回應至少10個的同步網頁請求。記憶體用於協定，系統緩衝器，應用程式都計算在內。因此設備驅動使用的記憶體將增加以上的數字.

15 API 參考手冊

15.1 資料類型

以下是LWIP API用到的兩種資料類型：
netbuf,網路緩衝器
netconn,網路連接
每種類型都是一個C架構體的指針，由這個架構體的內部架構我們知道它不應該在應用程式中使用，API取代它提供了修改和提取必要資料域的功能函式。

15.1.1 Netbufs

Netbufs 是用於發送接受資料的緩衝器，在6.1小節仲介紹了netbuf和pbuf之間的內部關連。Netbufs可以當做pbufs容納分發的記憶體和引用的記憶體。分發的記憶體是專門分發給持有網路資料的RAM，然而引用的記憶體可能不是應用程式管理的RAM就是外部的ROM。引用的記憶體對發送資料是很有用的，它不能修改，比如靜態的網頁和圖片。

資料在netbuf裡面可以是不同大小的碎片塊，這意味著應用程式必須準備好接受零碎的數據。netbuf內部有一個指針指向netbuf裡面的一個碎片，netbuf_next() 和netbuf_first()兩個函式就是利用了這個指針。從網路中接受的Netbufs封包含了發送資料封包來來源的IP位址，端口號。然後可以用函式來提取了那些存在的值。

15.2 Buffer 函式

15.2.1 netbuf new()

摘要
struct netbuf * netbuf new(void)

描述
分發一個netbuf架構體，執行這裡並沒有分發buffer空間，只是建立架構體的開始，執行後，必須用netbuf_delete()來刪除netbuf。

15.2.2 netbuf delete()

摘要
void netbuf delete(struct netbuf *)

描述
刪除先前由netbuf_new()分發的netbuf架構體，由netbuf_alloc()分發的任何buffer也會同時釋放。

例子 這個例子顯示了使用netbufs的最基本架構。
```
int
main()
{
struct netbuf *buf;
buf = netbuf_new(); /*  建立一個新的netbuf */
netbuf_alloc(buf, 100); /*  分發100 bytes  作爲buffer */
struct netbuf * netbuf new(void)
void netbuf delete(struct netbuf *)
/*  使用netbuf */
/* [...] */
netbuf_delete(buf); /*  刪除netbuf */
}
```

15.2.3 netbuf alloc()

摘要
void * netbuf alloc(struct netbuf *buf, int size)

描述
以位元組爲單位爲netbuf buf分發buffer，這個函式返回一個指針指向被分發的記憶體，先前曾經分發給過netbuf buf的空間會重新分發。分發的記憶體以後可以用 netbuf_free() 函式來重新分發。因爲發送資料的時候會先發協定頭，所以這函式爲協定頭分發空間的時候也爲實際的數據分發了空間。

15.2.4 netbuf free()

摘要
int netbuf free(struct netbuf *buf)

描述
重新分發和netbuf buf關連的buffer空間，如果buffer空間已經爲netbuf buf分發過，這函

式將會不做任何操作。

15.2.5 netbuf ref()

摘要
int netbuf ref(struct netbuf *buf, void *data, int size)

描述
將外部記憶體器的指針和netbuf buf的資料指針關連起來。外部記憶體器的大小由參數size提供。先前分發給netbuf的記憶體空間被重新分發。在用netbuf_alloc()爲netbuf分發空間和用malloc()分發空間和用netbuf_ref()涉及它不同的是，爲協定頭分發的空間會讓處理和發送buffer的速度更快。

例子 這個例子顯示了netbuf_ref()函式的簡單用法。

```
int
main()
{
struct netbuf *buf;
char string[] = "一個字元串";
/*  建立一個新的netbuf */
buf = netbuf_new();
void * netbuf alloc(struct netbuf *buf, int size)
int netbuf free(struct netbuf *buf)
int netbuf ref(struct netbuf *buf, void *data, int size)
/*  關連字元串*/
netbuf_ref(buf, string, sizeof(string));
/*  使用netbuf */
/* [...] */
/*  重新分發netbuf */
netbuf_delete(buf);
}
```

15.2.6 netbuf len()

摘要
int netbuf len(struct netbuf *buf)

描述
返回netbuf buf中資料的總長度值，不管netbuf是不是碎片狀的。netbuf是碎片的時候，返回值和netbuf裡第一個碎片的大小值是不一樣的。

15.2.7 netbuf data()

摘要
int netbuf data(struct netbuf *buf, void **data, int *len)

描述
此函式一般用於獲得一個指針和netbuf 裡面資料塊的長度。變量data和len分別是指向資料的指針和指向資料的長度。如果netbuf是碎片狀的，函式將指針指向netbuf裡面的一個碎片。應用程式必須用netbuf_first()和netbuf_next()碎片處理函式來訪問netbuf裡所有的資料。看下面netbuf_next()的例子如何使用netbuf_data()。

15.2.8 netbuf next()

摘要
int netbuf next(struct netbuf *buf)

描述
這個函式更新了指向netbuf buf內部碎片資料的指針，讓指針指向netbuf裡下一個碎片。如果netbuf裡還有資料片段存在，那么返回值將>0 如果當前指針已經指向了最後一個資料片，那么返回值<0.

例子 這個例子顯示如何使用netbuf_next()函式。我們假設函式中的buf變量是一個netbuf資料類型。

```
/* [...] */
do {
char *data;
int len;
int netbuf len(struct netbuf *buf)
int netbuf data(struct netbuf *buf, void **data, int *len)
int netbuf next(struct netbuf *buf)
/*  獲得一個指向資料片段的指針*/
netbuf_data(buf, &data, &len);
/*  使用這個資料*/
do_something(data, len);
} while(netbuf_next(buf) >= 0);
/* [...] */
```

15.2.9 netbuf first()

摘要
void netbuf first(struct netbuf *buf)

描述
複位netbuf buf裡面指向資料片段的指針，讓它指向第一個資料片段。

15.2.10 netbuf copy()

摘要
void netbuf copy(struct netbuf *buf, void *data, int len)

描述
複製所有netbuf buf裡面的資料到data指針指向的記憶體空間，即使netbuf buf是碎片狀的。len參數是複製到data指向記憶體空間的上限值。

例子 本例介紹了netbuf_copy()的簡單用法，這裡，在堆堆疊裡分發給資料的是200位元組的空間。即使netbuf buf裡面的資料是超過了200位元組，也只能複製200個位元組給data指針。

```
void example_function(struct netbuf *buf)
{
char data[200];
netbuf_copy(buf, data, 200);
/*  使用資料*/
}
```

15.2.11 netbuf chain()

摘要
void netbuf chain(struct netbuf *head, struct netbuf *tail)

描述
把兩個netbufs的頭和尾相連起來，因此這個的資料的尾將變成下一個資料片段的頭。在這個函式被呼叫後，原來netbuf的尾部將被重新分發不能再使用。

15.2.12 netbuf fromaddr()

摘要
struct ip addr * netbuf fromaddr(struct netbuf *buf)

描述
返回netbuf buf接收來自主機的IP位址。如果並沒有從網路接收到netbuf，那么將返回一個未定義的值。呼叫netbuf_fromport()函式可以獲得遠程主機的端口號。

15.2.13 netbuf fromport()

摘要
unsigned short netbuf fromport(struct netbuf *buf)

描述
返回netbuf buf接受來自主機的端口號。如果並沒有從網路接收到netbuf，那么將返回一個未定義的值。呼叫netbuf_fromaddr()函式可以獲得遠程主機的IP位址。

16 網路連接函式

16.0.14 netconn new()

摘要
struct netconn * netconn new(enum netconn type type)

描述
建立一個新的連接資料結構。變量可以是根據是TCP 連接還是UDP 連接分別爲NETCONN_TCP 或NETCONN_UDP。在連接沒被確定前呼叫此函式將不能從網路發送資料。

16.0.15 netconn delete()

摘要
void netconn delete(struct netconn *conn)

描述
刪除連接netconn conn 。如果呼叫此函式的時候這個連接已經打開，那么結果就會關閉這個連接。

16.0.16 netconn type()

摘要
enum netconn type netconn type(struct netconn *conn)

描述
返回連接conn的類型。這個類型將和netconn_new()的變量一樣不是NETCONN_TCP就是NETCONN_UDP。

16.0.17 netconn peer()

摘要
int netconn peer(struct netconn *conn, struct ip addr **addr, unsigned short port)

描述
在網路連接建立後呼叫本函式將獲得遠程連接的IP位址和端口。參數addr和port是函式設置的結果參數。如果連接conn沒有和任何遠程主機連接，將返回一個未定義的結果。

16.0.18 netconn addr()

摘要
int netconn addr(struct netconn *conn struct ip addr **addr unsigned short port)

描述
這個函式功能是用來獲取連接conn的本地IP位址和端口號。

16.0.19 netconn bind()

摘要
int netconn bind(struct netconn *conn struct ip addr *addr unsigned short port)

描述
把連接conn和本地IP 位址addr 和TCP或者UDP 端口結合起來。如果addr是無效的，本地IP 位址由網路系統確定。

16.0.20 netconn connect()

摘要
int netconn connect(struct netconn *conn, struct ip addr *remote addr, unsigned short remote port)如果UDP是透過遠程連接的接收者發送UDP消息來給出remote_addr和remote_port，那么對於TCP來說，netconn_connect()透過遠程主機來打開連接。

16.0.21 netconn listen()

摘要
int netconn listen(struct netconn *conn)

描述
讓TCP連機conn進入TCP監聽狀態。

16.0.22 netconn accept()

摘要

struct netconn * netconn accept(struct netconn *conn)

描述
阻止進程直到接收到遠程主機發送TCP連接conn的連接請求。連接必須處在監聽狀態，所以netconn_listen()的呼叫優先級必須比netconn_accept()高。當一個連接被遠程主機建立時，一個新連接架構被返回。

例子 這個例子來描述如何在2000端口上打開TCP伺服器
```
int main()
{
struct netconn *conn, *newconn; /*創建一個連接架構體*/
conn = netconn_new(NETCONN_TCP);
/*在任何本地IP位址上賦值連接2000端口*/
netconn_bind(conn, NULL, 2000); /*  通知連接來監聽增加的連接請求*/
netconn_listen(conn);
/* block直到我們獲取新的連接*/
newconn = netconn_accept(conn);
/*  使用newconn */
process_connection(newconn);
/*分發兩個連接*/
netconn_delete(newconn);
netconn_delete(conn);
}
```

16.0.23 netconn recv()

摘要
Struct netbuf * netconn recv(struct netconn *conn)

描述
在等資料在連接conn上到達時終止進程。如果遠程主機已經關閉連接，返回NULL，否則返回的是在netbuf中接收到的資料。

示例：透過下面一個小例子來說明如何使用函式netconn_ recv()。我們假設在呼叫函數example_function()之前連接已經建立。
```
void
example_function(struct netconn *conn)
{
struct netbuf *buf;
/*  接收資料直到主機關閉連接*/
while((buf = netconn_recv(conn)) != NULL) {
```

```
do_something(buf);
}
/*  連接在另一端被關閉，所以我們這端也關閉連接*/
netconn_close(conn);
}
```

16.0.24 netconn write()

摘要

int netconn write(struct netconn *conn void *data int len unsigned int flags)

描述

這個功能只用於TCP 連接。它簽訂被TCP 連接conn的關於輸出隊列的資料指向的資料。鏡子給資料的長度。沒有對資料的長度的限制。當這被這個堆照顧時，這個功能不要求申請明確地分發bu &-#174;ers。旗參數有兩個可能的國家，象如下所示的那樣。

這個函式僅用於TCP連接。It puts the data pointed to by data on the output queue for the TCP connection conn. 資料的長度由len給出。這裡沒有對資料的長度做限定。這個函式不需要應用程式明確分發緩衝區，這由堆堆疊來解決。參數flags有兩種可能的狀態，如下面所示：

```
#define NETCONN_NOCOPY 0x00
#define NETCONN_COPY 0x01
```

當使用NETCONN_COPY時flag資料被複製到爲這個資料分發的內部緩衝區。這種情況下允許資料在呼叫後直接修改，但是在執行的時間和記憶體的使用上效率是很低的。如果使用flag NETCONN_NOCOPY 資料就不是被複製而是引用。資料在呼叫後是不允許修改的，因爲資料被放在連接的重發隊列中，並保存在那裡對於那些不確定時間的。（stay there for an indeterminate amount of time.）當要發送的資料是固定不變存在ROM裡時這種方法就非常有用。

如果有大量的資料需要更改的話，可以是用複製和不複製資料的結合，如下面的例子所示。例這個例子說明了函式netconn_write()的基本用法。在這裡那些可變的資料假設在程式執行後被修改，因此資料將被複製到內部緩衝區透過定義flag NETCONN_COPY。變量text封包含一字元串將不被修改，這樣的話就可以使用引用來代替複製。

```
int
main()
{
struct netconn *conn;
char data[10];
char text[] = "Static text";
int i;
/*建立連接conn */
/* [...] */
```

```
/*創建一些隨機資料*/
for(i = 0; i < 10; i++)
data[i] = i;
netconn_write(conn, data, 10, NETCONN_COPY);
netconn_write(conn, text, sizeof(text), NETCONN_NOCOPY);
/*可以修改的資料*/
for(i = 0; i < 10; i++)
data[i] = 10 - i;
/*記下連接take down the connection conn */
netconn_close(conn);
}
```

16.0.25 netconn send()

摘要

int netconn send(struct netconn *conn struct netbuf *buf)

描述

使用UDP連接conn發送資料到netbuf buf。在netbuf裡的資料不能太大，因爲沒有使用IP分段。資料不能超過流出網路界面的最大傳輸單元（MTU）。因爲當前的資料無法透過一個恰當的途徑保存，netbuf不能記憶體大於1000位元組的資料。

無校驗使得不論發送的資料非常小或是非常大，netbuf可能給出不確定的結果。

例這個例子說明如何透過IP位址10.10.0.1的遠程主機的UDP端口7000發送UDP資料報。

```
int
main()
{
struct netconn *conn;
struct netbuf *buf;
struct ip_addr addr;
char *data;
char text[] = "A static text";
int i;
/*  創建一個新的連接*/
conn = netconn_new(NETCONN_UDP);
/*  設置遠程主機的IP位址*/
addr.addr = htonl(0x0a000001);
/*  連接遠程主機*/
netconn_connect(conn, &addr, 7000);
/*  創建一新的netbuf */
buf = netbuf_new();
data = netbuf_alloc(buf, 10);
/*創建一些立即數*/
```

```
for(i = 0; i < 10; i++)
data[i] = i;
/*  發送立即數*/
netconn_send(conn, buf);
netbuf_ref(buf, text, sizeof(text));
/*  發送text */
netconn_send(conn, buf);
/*分發connection  和netbuf */
netconn_delete(conn);
netconn_delete(buf);
```

16.0.26 netconn close()

摘要

int netconn close(struct netconn *conn)

描述

關閉連接conn。

17 BSD socket 庫

這一段給出了一個簡單的BSD socket API使用LWIP API的應用。這個例子僅僅提供一個參考，不是爲實際程式使用的。僅是個示例無錯誤處理。而且這個應用例子也不支援BSD socket API的select()和poll()函式，因爲在LWIP API裡沒有任何函式可以在這個應用中使用。爲了應用這些函式，BSD socket不得不直接和LWIP堆堆疊通信而不使用API。

17.1 socket的表示方法

在BSD socket API裡套接字被表示爲普通的文件描述符，文件描述符是一個獨一無二的標識，表示文件或網路連接的整型數。在這個BSD socket API應用中，套接字由netconn架構體來表示。因爲BSD sockets表示符是整型數，所以變量netconn放在sockets[]數組裡，BSD socket標識索引放在數組中。

17.2 分發socket

17.2.1 socket()函式的呼叫

呼叫socket（）函式來分發一BSD套接字。函式socket（）的參數用來規定何中套接字的類型。考慮到這個socket API的應用僅於網路套接字有關，所以在這裡僅支援一種套接字類型。而且僅僅UDP (SOCK DGRAM)或TCP (SOCK STREAM)套接字可以使用。

int

```
socket(int domain, int type, int protocol)
{
struct netconn *conn;
int i;
/*  創建netconn */
switch(type) {
case SOCK_DGRAM:
conn = netconn_new(NETCONN_UDP);
break;
case SOCK_STREAM:
conn = netconn_new(NETCONN_TCP);
break;
}
/*尋找sockets[]表中的空元素*/
for(i = 0; i < sizeof(sockets); i++) {
if(sockets[i] == NULL) {
sockets[i] = conn;
return i;
} }
return -1;
}
```

17.3 連接安裝

BSD socket API 對設置連接的呼叫函式與最低程度的API的連接設置函式十分相似.那些呼叫函數的執行主要封包括將socket的整數表示轉化爲在最低程度的API中使用的連接抽象化.

17.3.1 bind()函式呼叫

bind()呼叫函式將BSD socket綁定到本地(本機)位址上.在呼叫bind()時,本地IP位址和端口號將被指定.bind()函式與在lwIP API中的netconn_bind()函式十分相似.

```
int
bind(int s, struct sockaddr *name, int namelen)
{
struct netconn *conn;
struct ip_addr *remote_addr;
unsigned short remote_port;
remote_addr = (struct ip_addr *)name->sin_addr;
remote_port = name->sin_port;
conn = sockets[s];
netconn_bind(conn, remote_addr, remote_port);
return 0;
```

```
}
```

17.3.2 connect()函式呼叫

函式connect()的應用和bind()函式一樣簡單。（見bind()函式）

```
int
connect(int s, struct sockaddr *name, int namelen)
{
struct netconn *conn;
struct ip_addr *remote_addr;
unsigned short remote_port;
remote_addr = (struct ip_addr *)name->sin_addr;
remote_port = name->sin_port;
conn = sockets[s];
netconn_connect(conn, remote_addr, remote_port);
return 0;
}
```

17.3.3 listen()函式呼叫

listen() 呼叫函式相當於lwIP API中的netconn_listen()函式,它只能在TCP連接時使用.唯一的不同是BSD socket API允許應用程式指明等待連接隊列的大小.這對lwIP是不可能的並且等待事件的參數被忽略了.

```
int
listen(int s, int backlog)
{
netconn_listen(sockets[s]);
return 0;
}
```

17.3.4 accept()函式呼叫

accept()呼叫函式被用來等待TCP socket口上的輸入連接.事前,這個TCP socket口透過呼叫listen()已經被設置成監聽狀態.對accept()呼叫一直被阻塞,直到與遠程主機建立連接.監聽的自變量是結果參數,這些參數透過呼叫accept()來設置.他們充滿了遠程主機的位址.當新的連接建立時,lwIP函式netconn accept()因爲新的連接將返回連接句柄.在遠程主機的IP位址和端口被裝滿後,新的socket口的標識符被分發和返回.

```
int
accept(int s, struct sockaddr *addr, int *addrlen)
{
struct netconn *conn, *newconn;
struct ip_addr *addr;
unsigned short port;
int i;
```

```
conn = sockets[s];
newconn = netconn_accept(conn);
/*取得遠程主機的IP位址和端口號*/
netconn_peer(conn, &addr, &port);
addr->sin_addr = *addr;
addr->sin_port = port;
/*分發新的套接字標識*/
for(i = 0; i < sizeof(sockets); i++) {
if(sockets[i] == NULL) {
sockets[i] = newconn;
return i;
} }
return -1;
}
```

17.4 發送和接收資料

17.4.1 send()函式呼叫

在BSD socket API中,send()呼叫函式在UDP 和TCP兩種連接中被用來發送資料.在呼叫send()前,資料接收器必須被設置成正在使用connect().對UDP期間,send()呼叫函式類似lwIP API中的netconn send()函式,但是因爲lwIP API需要應用程式明確地分發緩衝區,所以一個緩衝區必須在send()呼叫函式裡分發和釋放.因此,先分發緩衝區再把資料複製進緩衝區.

lwIP API中的netconn send()函式不能在TCP連接中使用,因此在TCP連接中使用netconn write()來執行send()功能.在BSD socket API中,應用程式在呼叫send()後可以直接修改發送的資料,因此NETCONN COPY 標誌位被傳遞給netconn write()那樣資料被裝入了堆堆疊的內部緩衝區.

```
int
send(int s, void *data, int size, unsigned int flags)
{
struct netconn *conn;
struct netbuf *buf;
conn = sockets[s];
switch(netconn_type(conn)) {
case NETCONN_UDP:
/*創建一緩衝區*/
buf = netbuf_new();
/*  使得緩衝區指針指向要發送的資料*/
netbuf_ref(buf, data, size);
/*發送資料*/
netconn_send(sock->conn.udp, buf);
```

```
/*分發緩衝區*/
netbuf_delete(buf);
break;
case NETCONN_TCP:
netconn_write(conn, data, size, NETCONN_COPY);
break;
}
return size;
}
```

17.4.2 sendto() 和sendmsg()函式呼叫

sendto() 和sendmsg()呼叫函式與send()呼叫函式類似,但是在參數呼叫中他們允許應用程式指定資料接收器.同樣,sendto() 和sendmsg()僅能在UDP連接中使用.實現這功能要使用etconn connect()來設置資料封包接收器.如果以前socket口被連接,因此必須重設遠程IP位址和端口號.不封包括sendmsg()的執行不封包括.

```
int
sendto(int s, void *data, int size, unsigned int flags,
struct sockaddr *to, int tolen)
{
struct netconn *conn;
struct ip_addr *remote_addr, *addr;
unsigned short remote_port, port;
int ret;
conn = sockets[s];
/*當前連接建立獲取peer */
netconn_peer(conn, &addr, &port);
remote_addr = (struct ip_addr *)to->sin_addr;
remote_port = to->sin_port;
netconn_connect(conn, remote_addr, remote_port);
ret = send(s, data, size, flags);
/*複位遠程連接的位址和端口號*/
netconn_connect(conn, addr, port);
}
```

17.4.3 write()函式呼叫

在BSD socket API中,write()呼叫函式透過連接來發送資料並且能在UDP 和TCP連接中使用.對TCP連接來說,這個函式直接映射到lwIP API函式netconn write().對UDP連接來說,BSDsocket函式write()等價於send()函式.

```
int
write(int s, void *data, int size)
{
```

```
struct netconn *conn;
conn = sockets[s];
switch(netconn_type(conn)) {
case NETCONN_UDP:
send(s, data, size, 0);
break;
case NETCONN_TCP:
netconn_write(conn, data, size, NETCONN_COPY);
break;
}
return size;
}
```

17.4.4 recv()和read()函式呼叫

在BSD socket API,透過函式recv() and read()呼叫來連接socket接收資料.可以用在TCP和UDP連接上.透過呼叫函式recv()來傳遞很多flags.在這裡這些都用不著,因爲flags參數是被忽略的.如果接收到的消息大於提供的記憶體區間,多於的資料將自動丟棄.

```
int
recv(int s, void *mem, int len, unsigned int flags)
{
struct netconn *conn;
struct netbuf *buf;
int buflen;
conn = sockets[s];
buf = netconn_recv(conn);
buflen = netbuf_len(buf);
/*  複製接收緩衝區的內容到提供的記憶體指針men指向的區域*/
netbuf_copy(buf, mem, len);
netbuf_delete(buf);
/*  如果接收資料長度大於len,資料被丟棄並返回len,否則返回接收到的實際資料的長度.*/
if(len > buflen) {
return buflen;
} else {
return len;
} }
int
read(int s, void *mem, int len)
{
return recv(s, mem, len, 0);
```

}

17.4.5 The recvfrom() and recvmsg() calls

函式recvfrom()和recvmsg()的呼叫和recv()類似,區別在於前者可以透過呼叫函式獲取資料發送者的IP位址和端口號.recvmsg()的應用這裡沒有給出.

```
int
recvfrom(int s, void *mem, int len, unsigned int flags,
struct sockaddr *from, int *fromlen)
{
struct netconn *conn;
struct netbuf *buf;
struct ip_addr *addr;
unsigned short port;
int buflen;
conn = sockets[s];
buf = netconn_recv(conn);
buflen = netbuf_len(conn);
/*複製接收緩衝區的內容到提供的記憶體指針men指向的區域*/
netbuf_copy(buf, mem, len);
addr = netbuf_fromaddr(buf);
port = netbuf_fromport(buf);
from->sin_addr = *addr;
from->sin_port = port;
*fromlen = sizeof(struct sockaddr);
netbuf_delete(buf);
/*如果接收資料長度大於len,資料被丟棄並返回len,否則返回接收到的實際資料的長度.*/
if(len > buflen) {
return buflen;
} else {
return len;
} }
```