第八章 TCP Socket 程式介面

8-1 Socket 簡介

由前面幾章的介紹，我們可以瞭解 Internet 網路架構及相關通訊協定，由本章開始將介紹如何在 Internt 網路上開發應用程式的基本概念。一般在 TCP/IP 環境下開發應用程式有兩個基本介面程式：**『插座』(Sockets)** 和 **『傳輸層介面』(Transport Layer Interface, TLI)**。但在 Internet 網路上的應用程式都是以 Socket 介面佔大部分，我們就以 Socket 為範例來介紹通訊程式的開發技巧，如果讀者對於 TLI 有興趣的話，請參考其他書籍，本書限於篇幅不另介紹。第十一章將以 Socket 為基礎，更進一步介紹較高階的程式介面 – 遠端程序呼叫 (Remote Procedure Call, RPC)。我們有了這些網路應用程式的基本概念後，接著在第四部分更進一步的介紹網路應用系統及其相關之通訊協定。

8-1-1 Socket 與 TLI

TLI 和 Socket 皆是 **『應用程式介面』(Application Program Interface, API)**，使用者可透過 API 發展網路應用程式，但兩者有許多不同的地方。TLI 介面最早是在 Unix System V 3.0 版中出現，其目的是作為 OSI 模式中傳輸層介面使用；而 Socket 由 BSD(Berkeley Software Distribution) 所發展出來，又稱為 **『Berkeley Socket』**。TLI 介面和 Socket 介面最大不同點，在於 Socket 提供程式庫集的連結方式，稱之為 Socket Library，如 read()，write() 等功能呼叫，這些功能呼叫的特性就如同 Unix 的系統呼叫 (System Call) 一樣；而 TLI 僅提供層次間的介面呼叫。圖 8-1 (a) 為 Unix System V 之 TLI；(b) 為 BSD Socket 介面的系統模組。

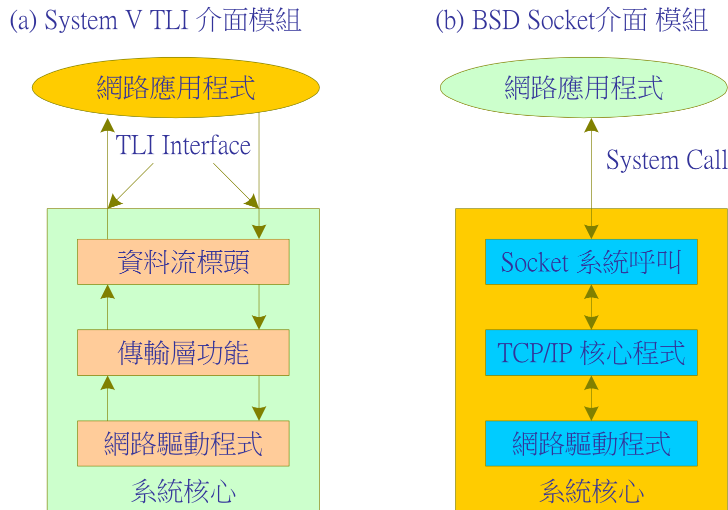

圖 8-1 TLI 與 Socket 介面的系統模組

由作業系統觀念而言，Socket 介面所提供的 read()、write() 都是屬於系統核心（Kernel）內部的呼叫程式，這是因為一般 Unix/Linux 作業系統都將 TCP/IP 功能整合在系統核心內。因此，利用 Socket 介面來編寫網路應用程式，和一般編寫應用程式（非網路環境）方式，並沒有太大的區別，這一點讓使用者非常容易發展網路程式，這就是 Socket 介面最大的特點。目前許多作業系統（如 Windows 2000 或 Novell）都將 TCP/IP 核心程式載入系統核心內部，也依然可利用 Socket 介面發展程式，此類作業系統大多針對網路提出特殊功能，通常稱之為『網路作業系統』(Network Operating System, NOS)。而 TLI 僅是一個介面，使用者必須將 TLI Library 和網路程式連結(Link)之後再產生執行程式，在執行時是透過 Unix 的資料流輸出入介面（Stream I/O）和傳輸層溝通。也就是說，TLI Library 是在執行中再呼叫核心程式，處理有關網路運作，而並非整合在核心程式內。

但在 Unix/Linux 作業系統之下，雖然 TLI 和 Socket 介面在實現方式上有些差異，但目的是一致的，同樣可開發各式各樣的網路應用程式，隨著系統工具或使用者習慣可選擇其中一種介面來應用。但目前許多安全性介面（如 Secret Socket Layer, SSL）都在 Socket 上發展，既然 Socket 介面發展程式已漸成趨勢，接下來我們就以 Socket 介面為範例來介紹。

8-1-2 何謂 Socket

如果僅以 Socket 字元來翻譯是『**插座**』的意思，這個名詞會讓人非常訝異，但話說回來，它的功能的確像電話『**插座**』一樣。如以電話系統而言，只要將電話的插座設定好某一號碼，任何一部電話都可透過這個插座和其它電話通訊。對使用者而言，不用理會電話系統是如何撥接、或是如

何路由選擇到達目的，這些功能完全由電話公司負責。網路上的 Socket 也如同電話的插座一樣，任何一個 Socket 都給予一個特殊號碼(IP number + TCP port)，使用者之間只要記住對方的 Socket 號碼，便可以直接通訊，而不用考慮到底是經過何種網路、或主機放在什麼地方，這些尋找主機的工作是網路提供者所必須負責的。因此，對於 Socket 我們可以定義如下：

『Socket 就是一個網路上的通訊端點，使用者或應用程式只要連接到 Socket 便可以和網路上任何一個通訊端點連線，Socket 之間通訊就如同作業系統內程序（Process）之間通訊一樣。』

圖 8-2 為 Socket 之間的連線方式。透過 Socket 之間通訊，使用者不用理會網路實體架構，且 Socket 介面呼叫也如同作業系統的系統呼叫，對一般系統發展者而言，發展網路應用程式如同一般程式的容易，因此，稱之為**『插座』（Socket）**是最為適合。Windows 作業系統也提供 Socket 的介面程式，稱之為 Windows Socket（Winsock）。

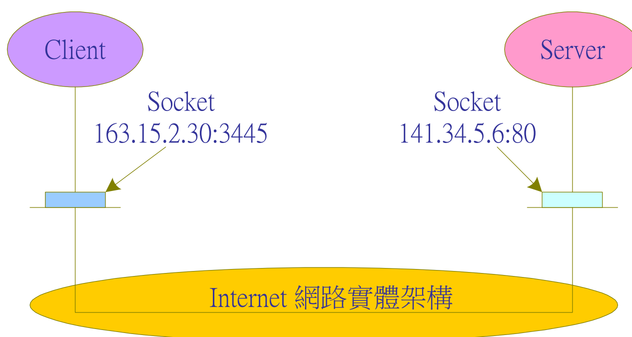

圖 8-2 Socket 的連接方式

8-2 Socket 基本功能

Socket 介面提供六個基本功能，來達成通訊連線的目的：

(A) 開啟插座（Create a socket）

我們可以用 socket() 呼叫程式，來產生一個通訊端點。在呼叫 socket() 程序時，必須給予適當的參數作為規劃通訊通道的模式，通道參數如下：

- **Address Family：**表示通道的位址模式，例如：Internet 或 Unix 通訊模式。
- **Type of Service：**任何一種通道模式都有其特殊的服務模式，一般可區分為電報傳輸（Datagram）或虛擬電路（Virtual Circuit）服務兩大類。

- **Protocol：**此通訊端點所使用的通訊協定，可區分為 TCP、UDP 或 IP。

(B) 連結通訊位址（Bind an address）

當開啟一個 Socket 後並不能直接通訊，就好像有一個電話機，但沒有電話號碼一樣，必須將電話機連結到電話交換機上，並給予一個電話號碼。bind() 程序就是將 Socket 連結到通訊端點上，並給予一個通訊埠口號碼，然而通訊號碼是由**『IP Address + Port Number』**所構成，有了這個號碼就成為 Internet 網路上一個成員，就好像電話機已連上一個電話號碼一樣，利用這個號碼就可以和其它成員通訊。圖 8-3 表示將 Socket 植入通訊埠口號碼的示意圖，通訊雙方都設定到各自的通訊號碼，雙方便可依照這個號碼來通訊，而不用理會網路實際的連接型態。

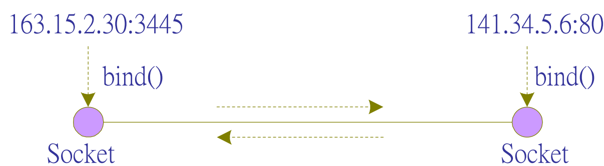

圖 8-3 Socket 埠口號碼

(C) 連結到對方插座（Connect to another socket）

當 Socket 連結到位址後，便可以和遠端 Socket 連線，來建構一個完整的通訊連線。通訊模式有兩種連線方式，一者為虛擬電路（Virtual Circuit）方式，通訊雙方需事先建立連線，才可以互相通訊，就好像必須撥接通電話，雙方才可以對話。連線建立後，雙方便依照此連線傳輸資料，資料傳送中無需再建立連線；另一者為電報傳輸（Datagram），雙方 Socket 並未建立連線，便直接將訊息傳送給對方，它的動作就好像將訊息傳送到某一信箱內，每一信息（或稱 Datagram）都必須註明對方的位址，也就是說，每一信息都必須自行建立通訊連線，再將訊息傳送過去，傳送完之後，該連線便自動消失。

(D) 接受對方端點連線（Accept a socket connection）

接受對方連線的情況只會發生在虛擬電路方式。當 Socket 連結到網路位址後，便可選擇進入聆聽狀態（Listen State），以等待遠端 Socket 要求連線。當 Socket 接收到要求連線訊息後，可選擇是否接受連線（Accept()）。

(E) 傳送資料（Transfer data）

Socket 之間傳送資料，就好像一般程式將資料寫入磁碟機或緩衝器一樣的簡單，依照連結方式有不同的介面程式，如虛擬電路連接方式有：

- read() 和 write() 系統呼叫是作為一般資料的傳送與接收使用。
- send() 和 recv() 提供比 read()/write() 較多功能的系統呼叫。一般使用於較緊急的資料傳輸，這兩個系統呼叫所傳送的資料，會超越一般資料的佇列排序，而優先傳送或接收。

如果採用電報傳輸方式有：

- sendto() 和 recvfrom() 系統呼叫是做一般資料傳送與接收使用。

為了達到交談式的連線傳輸，不論虛擬電路或電報傳輸都可用下列兩個系統呼叫來傳輸資料：

- sendmsg() 和 recvmsg()。

(F) 停止插座操作（Shut down socket）

當執行 close() 系統呼叫停止 Socket 動作之後，將禁止所有資料傳輸。

8-3 Socket 連接方式

利用 Socket 介面發展網路應用程式有：虛擬電路和電報傳輸兩個主要通訊模式，以下我們以 Client/Server 架構分別介紹兩種通訊模式。

8-3-1 虛擬電路模式

『虛擬電路』（Virtual Circuit）模式表示通訊雙方傳輸資料之前，必須先建立通訊鏈路，因此所採用的傳輸層為連接導向的 TCP 協定。圖 8-4 是利用 Socket 程式庫，使用 TCP 協定建立的 Client/Sever 模式之基本架構，圖中 Server 端，首先呼叫 socket() 開啟一個通訊端點，接下來以 bind() 設定連接之傳輸埠口，並以 listen() 功能呼叫設定該 Socket 通訊端點為聆聽狀態（Listen Mode），準備接收對方要求連線，最後呼叫 accept() 等待 Client 端的連線。同樣的，Client 端也以 socket() 呼叫建立通訊端點，並以 bind() 連接到傳輸埠口，此時 Client 便可用 connect() 功能呼叫要求連接到 Server 端。雙方連線成功之後，便可以用 read() 和 write() 來互相傳輸資料，最後以 close() 結束此連線。

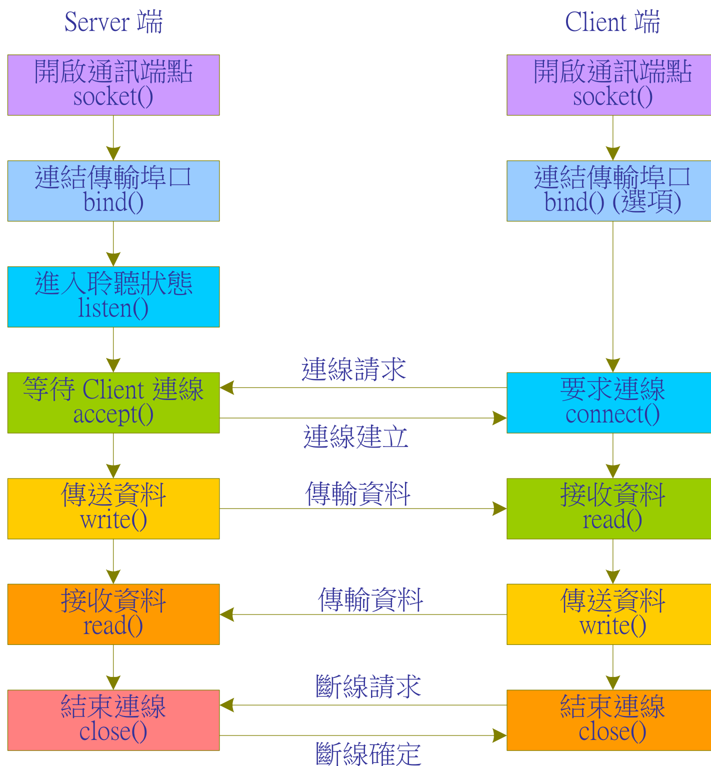

圖 8-4　Socket 的虛擬電路模式

在虛擬電路中有：同步（Synchronous）和非同步（Asynchronous）兩種傳輸模式。同步傳輸模式表示執行 Socket 程式後，還未得到回應時，會繼續等待一直到有回應才會回主程式，因此又稱為**『阻斷模式』(Blocking Mode)**。譬如，Server 端執行 accept() 程式，會一直等待到有連線到達，才會回應給主程式；而非同步模式表示執行 Socket 程式會即時回應給主程式，不論是否取得資料，又稱為**『非阻斷模式』(Nonblocking Model)**。

另一種情況，在同步傳輸模式中執行 read() 程式，如果接收緩衝器上有資料，便會直接讀取並返回主程式，但如果緩衝器上沒有資料，則會一直等待到連線對方傳送資料過來，才會返回主程式。非同步傳輸模式則不然，不論緩衝器上是否有資料可以讀取，都會即時返回主程式。由此可以見，同步傳輸只適合一對一的通訊端點傳輸，但大部分的 Client/Server 架構，Server 端必須能同時接受多個 Client 端的要求連線，同步傳輸便不適合。一般開啟連接導向的通訊端點，內定值都是同步傳輸模式，如果通訊端點欲能夠接受多點連接（一般都是 Server 端），必須將 Socket 設定為非同步傳輸模式。又從另一觀點來看，執行 read() 程式時，是由緩衝器上讀取資料，Socket ID 也如同一般檔案識別碼一樣，因此我們可以利用一般檔案控制程式 fcntl() 來設定。因此設定 Socket 為非同步傳輸模式的方法非常簡單，開啟 Socket 後，以 fcntl() 程式將 FNDELAY 旗號加入到 Socket ID（sd）內即可，程式格式如下：

```
fcntl(sd, F_SETFL, FNDELAY);
```

如欲將 Socket 由非同步傳輸模式改為同步傳輸模式，也就是將旗號 FNDELAY 消除，其程式格式如下：

```
int flags;


flags = fcntl(sd, F_GETFL, 0);
fcntl(sd, F_SETFL, (flags & ~FNDELAY));
```

參數 sd 為 Socket ID，第一個 fcntl() 程式是讀取 Socket 上的旗號值，而第二個 fcntl() 是作為取消 FNDELAY 旗號使用。

當 Socket 被設定成非同步傳輸模式之後，必須能夠隨時接收到資料的到達，以下有兩種方法：

(1) 週期性的執行 read() 功能呼叫，隨時準備接收及時到達之資料。

(2) 設定 Socket 具有發送訊息的功能，譬如，當一般資料到達時，會發送 SIGIO 訊號給主程式，再由主程式執行 read() 程式來讀取，或當快速資料（Expedited Data）到達時，會發送給 SIGURG 給主程式。

8-3-2 電報傳輸模式

圖 8-5 為建立在 UDP 協定上的**『電報傳輸』(Datagram)**模式，也是 Client/Server 的基本架構。Server 和 Client 端分別以 socket() 功能呼叫開啟通訊端點，並以 bind() 連結到傳輸埠口位址，其中 Client 端是否執行 bind() 為選項項目。雙方以 recvfrom() 和 sendto() 互相傳送資料。

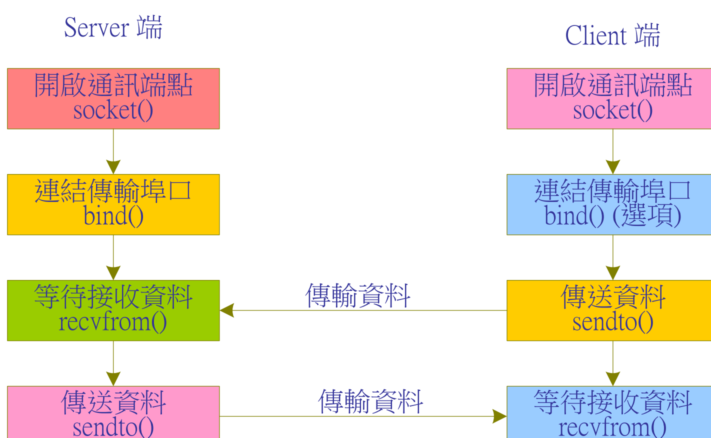

圖 8-5 Socket 的電報傳輸模式

由圖 8-5 中表示，電報傳輸的預定值也是同步傳輸模式，也就是說，當某一方執行 recvfrom() 時，必需等待到收到對方傳送過來資料，才會返回到主程式，否則會一直等待著(Blocking Model)。如同虛擬電路一樣，我們可以利用 fcntl() 將其設定成非同步傳輸模式，程式格式如下：

```
fcntl(sd, F_SETFL, FNDELAY)
```

參數 sd 是 Socket 的識別碼，可以是 Server 端或 Client 端。取消 FNDELAY 也如同虛擬電路一樣，不再另述。

8-4 Socket 傳輸位址

早期 BSD 制定 Socket 介面時，希望 Socket 能連結到不同的傳輸提供者(Transport Provider)，但各種傳輸提供者都有其特殊的定址方式，因此，Socket 必須提供一個標準的傳輸位址模式來轉換不同的位址格式。首先我們來探討傳輸位址必須具有哪些內容，一般當我們用 Socket 程式庫時會有如下列步驟：

(1) 將 Socket 連結到一個傳輸位址。

(2) 連接到遠端 Socket。

(3) 傳輸資料。

此時 Socket 程式庫至少必須攜帶兩個參數：一者為連結到傳輸位址的指標；另一者為包含多少個緩衝器的資料。目前在許多 Unix/Linux 作業系統裡，為了能結合多種不同的傳輸提供者（也就是傳輸層），也包含許多傳輸位址格式，以下列出 Internet 網路較常使用的三種位址格式：

(1) sockaddr：Unix 作業系統格式（AF_UNIX）。

(2) sockaddr_in：Internet 網路格式（AF_INET）。

(3) sockaddr_un：本機迴授位址（Loopback）格式（AF_UNIX）

以下分別介紹之。

8-4-1 Unix 位址格式 - sockaddr

在 Unix/Linux 系統上，BSD 針對系統核心制定一個網路位址格式，它是以 16 位元之 Socket 位址來表示，又以 sockaddr 資料型態來表示 Socket 位址，其資料結構如下：

```
struct sockaddr {
      u_short      sa_family;
      char          sa_data[14];
}
```

- **sa_family**：表示 Socket 的系列家族，為 sockaddr 的前面兩個位元組。
- **sa_data[14]**：表示所描述 Socket 位址的資料。在 BSD 核心系統的操作下，sa_data 包含路由表、介面位址，以及核心所描述特殊表格的資料。

8-4-2 Internet 位址格式 - sockaddr_in

針對 Internet 網路上傳輸，BSD 也制定一個 sockaddr_in 資料結構，來描述網路之間的位址格式，其資料結構如下：

```
struct sockaddr_in {
         u_short                sin_family;
         u_short                sin_port;
         struct    in_addr    sin_addr;
         char                   sin_zero[8];
};
```

各欄位功能如下：

- **sin_family**：表示該 Socket 的通訊協定家族系列，可選擇：AF_UNIX（Unix 作業系統）、AF_INET（Internet 網路）、AF_NS 或 AF_IMPLINK 等。
- **sin_port**：表示該 Socket 所連接的埠口位址，如通訊家族為 AF_INET，則表示 TCP/UDP 埠口號碼。
- **sin_addr**：表示該 Socket 所連接之 IP 位址。對於 32 位元 IP 的表示方法有三種：(1) 用 4 位元組將 IP 位址分割為 4 個十進位之數字表示（如 163.15.2.62）; (2) 區分為兩個群組表示（網路位址 + 主機位址）; (3) 直接以一個數字表示。因此，用 in_addr 資料結構來宣告位址格式方式：

```
struct in_addr {
```

```
        union {
                struct { u_char s_b1, s_b2, s_b3, s_b4; } S_un_b;
                struct { u_short s_w1, s_w2; } S_un_w;
                u_long s_addr;
          } ;
};
```

- **sin_zero**：目前未使用，一般都為 0。

8-4-3 Loopback 位址格式 - sockaddr_un

在 Unix/Linux 系統下，如果使用迴授介面（Loopback）開發 Socket 應用程式時，所使用的位址格式為 sockaddr_un，其定義如下：

```
struct sockaddr_un {
        u_short         sun_family;
        char            sun_path[108];
};
```

各欄位功能如下：

- **sun_family**：通訊協定家族。在這種情況下必須使用 AF_UNIX，而其值為 1。
- **sun_path**：路徑位址（Pathname）。在這裡為 Unix 的檔案名稱，如果該檔案存在，則這個位址被使用到，否則不被使用，而等待連結到 Socket 位址。

8-5 Socket 庫存函數

Socket 提供一系列的程式庫讓使用者來編寫網路應用程式，本節將依照 RedHat Linux（BSD 標準）來介紹**『Socket 庫存函數』（Socket Library）**與相關程式的呼叫方式，至於其它作業系統也幾乎都是一樣。

8-5-1 Socket 庫存函數彙集

我們可以透過 Socket 庫存函數來處理兩個傳輸提供者之間的交談，它所提供的功能呼叫彙集如表 8-1。

表 8-1 Socket 庫存函數

開啟 Socket（Create a Socket）	
socket()	開啟一個 Socket，並回應一個檔案描述子
socketpair()	開啟兩個 Socket 且連結它們，並回應兩個檔案描述子
定名 Socket（Names a Socket）	
bind()	將 Socket 定名，表示連結到傳輸提供者之位址上。
連結 Socket（Connects to a Socket）	
listen()	將 Socket 設定進入『聆聽』狀態。
accept()	Socket 等待遠端連接訊號到達，並接受連線。
傳輸資料（Transfer Data）	
send()/write()	透過連線傳送訊息。
sendto()	傳送分連接訊息（Datagram），必須描述對方 Socket 位址。
sendmsg()	傳送交談式訊息，必須描述對方 Socket 位址。
recv()/read()	透過連線接收訊息。
recvfrom()	接收非連接訊息（Datagram），包含傳送端 Socket 位址。
recvmsg()	接收交談式訊息，包含傳送端 Socket 位址。
停止 Socket（Shuts Down a Socket）	
shutdown()	停止 Socket 工作，並禁止所有 Socket 之連線及傳輸動作。
Close()	同上

管理 Socket 程式集	
getsockname()	取得本身 Socket 的名稱。
getpeername()	取得對方 Socket 的名稱。
setsockopt()	設定 Socket 某些參數。
getsockopt()	取得 Socket 參數值。

以下針對較常使用的庫存函數加以介紹。

8-5-2 開啟 Socket

socket() 功能呼叫是用來產生一個通訊端點，也就是向系統註冊，通知系統要建立一個通訊端點，其程式格式如下：

```
#include <sys/types.h>
#include <sys/socket.h>

int domain;
int type;
int protocol;
int fd;

fd = socket(domain, type, protocol);
```

socket() 如同一般檔案系統的 open() 系統呼叫一樣。當 socket() 執行正常時，會回應一個整數(fd)，一般稱之為**『檔案描述子』（File Description, fd）**，表示該 Socket 的識別值。如果檔案描述子大於 0 則表示開啟正常，並針對該檔案描述子作處理動作（connect()、read() 等）。在一個應用程式裡，可經過多個 socket() 呼叫，來產生多個通訊端點，並建立多個通訊通道，這些通訊端點就以不同的檔案描述值來區分。

當呼叫 socket() 系統程式時，必須給予相對應之參數，說明如下：

(A) domain 參數

參數 domain 或稱為 family，是用來選擇使用哪一種通訊協定的家族系列，domain 可選擇下列之一：

- **AF_UNIX**：Unix Internet Protocol。此通訊家族並不是真正的網路通訊協定，而是用來作 Unix 作業系統中，各程序（Process）之間的通訊使用。一般使用在回授（Loopback）傳輸提供者，而其應用在主機內程序之間的通訊。
- **AF_INET**：Internet Protocol。此為 TCP/IP 的 Internet 通訊協定，傳輸提供者可能是 TCP 或 UDP，也是本章討論的重點。
- **AF_NS**：Xerox NS Protocol。此為 Xerox 公司發展的通訊協定。
- **AF_IMPLINK**：Interface Message Protocol。此為一種智慧型的分封交換節點協定，這些節點都是使用點對點的連接方式，一般使用在租用電話線路來作資料傳輸使用。(不在本章討論範圍)

其中『**AF_**』代表 Address Family，有些系統使用『**PF_**』(**Protocol Family**)，兩者是相通的。domain 參數也如同 sockadd_in 資料結構中的 sin_family 參數一樣。

(B) type 參數

參數 type 是設定該 Socket 的類型，可選擇下列類型之一：

- **SOCK_STREAM**：Stream Socket。傳輸提供者提供一個虛擬電路服務（TCP）。
- **SOCK_DGRAM**：Datagram Socket。提供電報傳輸服務（UDP）。
- **SOCK_RAW**：Raw Socket。通訊協定型態在傳輸層之下，譬如，在 AF_INET（傳輸層為 TCP 或 UDP）模式，SOCK_RAW 的通訊協定可以是 IP（Internet Protocol）或 ICMP（Internet Control Message Protocol）。
- **SOCK_SEQPACKET**：Sequenced Packet Socket。提供虛擬電路（TCP）並附有維護訊息的功能。

(C) protocol 參數

參數 protocol 是在某一個 domain 之下，選擇所要哪一種協定。例如選定 AF_INET domain 系列時，所使用的協定可以是 TCP、UDP 或 IP 中的一種。但當設定 domain 和 type 值時，對於所使用的協定大多已經指定完成，因此 protocol 的值一般都設定為 0。但有一特殊情況，如果 Socket 的型態是 SOCK_RAW 時，必須在參數中指定它的上層協定為 TCP、UDP、IP 或 ICMP。

另一個系統呼叫 socketpair() 是用來開啟兩個 Socket，並建立它們之間的連線。經過 socketpair() 呼叫所產生的兩個 Socket 必定是在同一系統上，好像一般 Unix 上的管道（Pipe）一樣，因此，domain 只能設定為 AF_UNIX，否則會發生錯誤。socketpair() 的語法如下：

```
#include <sys/types.h>
#include <sys/socket.h>

int domain, type, protocol, status, fd_array[2];

status = socketpair(domain, type, protocol, fd_array);
```

如果系統執行正常，status 會回應一個大於 0 的數值，而 fd_array 回應兩個檔案描述子來代表 Socket。

8-5-3 定址 Socket

產生一個 Socket 之後，必須針對該 Socket 定址，才會產生作用。定址的目的是將 Socket 連結到傳輸埠口上，Socket 才會有適當的通訊位址，因此稱之為**『定址』(Addressing)**。定址 Socket 的功能呼叫是 bind()，它的基本做法是將已開啟的 Socket 描述子，再加上欲連接的 IP 位址和傳輸埠口，建構一個資料結構，再呼叫系統核心連接傳輸埠口。所建立的資料結構依照通訊協定方式有所不同：

- **AF_UNIX**：由 <sys/socket.h> 包含檔中定義 sockaddr 資料結構，其型態如 8-4-1 節所示。
- **AF_INET**：則在 <netinet/in.h> 包含檔中定義兩個資料結構：socketaddr_in 和 in_addr，其格式如 8-4-2 節所示。

- **AF_NS**：則在 <nstns/ns.h> 包含檔中描述 ns_addr 與 sockaddr_ns。AF_NS 不在本書討論範圍，因而不再另述。

我們以 AF_INET 為範例，bind() 呼叫程序如下：

```
#include <sys/types.h>
#include <netinet/in.h>
int fd;
struct sockaddr_in    *addressp;
int addrlen;
int status;


status = bind(fd, addressp, addrlen);
```

在 bind() 程式中，fd 為所欲連結之 Socket 的描述子，addressp 包含有關 Socket 欲連接的環境參數，如 IP 位址及 TCP 埠口等；addrlen 表示 addressp 資料的長度，以位元組計算。如果呼叫成功，則 status 會回應一個大於 0 的整數，其中 sockadd_in 資料結構內 IP 位址和埠口號碼的位元次序，與網路位元組的次序相反，一般必須經過位元組次序轉換（ntonl() 等函數）。至於如何建構 sockaddr_in 的內容，請參考 8-6 範例說明。

8-5-4 接受連線請求

一般網路應用環境都以 Client/Server 架構為大宗，Server 端的 Socket 經過定址後，必須進入聆聽（Listen）狀態，準備聆聽來自 Client 端的連線要求，但並未開始接受連線請求。Server 端再進入接收狀態（Accept），隨時接收 Client 端的連線訊息。當 Server 端執行 accept() 系統呼叫時，會進入等待狀況（Wait State），一直到接收到連線請求為止，因此接受對方連線有兩個功能呼叫為：listen() 和 accept()。

(A) listen()：設定 Socket 進入 Listen 狀態

```
int fd;
int qlen;
int status;
```

```
status = listen(fd, qlen);
```

listen() 設定 Socket fd 進入聆聽狀態，其中 sqlen 表示 Server 在執行 accept() 功能呼叫之前，系統所能佇列（Queue）Client 端要求的連線數目，如果執行正常，則 status 回應 0，否則回應 -1。listen() 是屬於 Server 端的功能呼叫，而且只適用於 SOCK_STREM 與 SOCK_SEQPACKET 類別（連接導向服務）的 Socket 上。

listen() 通常在 socket()、bind() 功能呼叫之後執行，其後緊接著 accept() 功能呼叫。雖然 listen() 只設定 Socket 成為聆聽狀態，但是系統執行此一功能呼叫時，已將下層的 TCP 狀態設定為接收狀態，當有連線要求進來時，下層的 TCP/IP 程式依然會接收連線，而將進來的連線要求排入佇列內，等待 accept() 系統呼叫來索取訊息。另一方面，上層的應用程式也必須執行 accept() 後，才會知道是否有連線進來。

(B) accept()：接受對方連線要求

```
#include <sys/types.h>
#include <sys/socket.h>

int fd;
struct sockaddr *peeradr;
int peeradrlen;
int newfd;

newfd = accept(fd, peeradr, &peeradrlen);
```

通常 accept() 是在 listen() 之後執行的，以便接收來自 Client 端的連線請求，fd 的值和 listen() 中所使用的 fd 值相同。

如果在同步傳輸模式（Synchronous），則 accept() 會等待到連線到達。當連線到達時，accept() 會再開啟一個新的 Socket(newfd)，新的 Socket 繼承原來 Socket(fd)的所有特性(也就是 newfd 可當作 fd 的 Child Socket)，爾後所有連線的處理動作都由新 Socket(newfd)負責，而原來 Socket（fd）則回到聆聽狀態，準備接受下一個連線請求。

當 accept() 執行成功時，會回傳 Client 端的 Socket 位址，以標明哪一個 Socket 要求連線。Server 端必須提供一個空的 Socket 資料結構（peeradr）來承接該位址，而其依照不同通訊型態有各自的資料結構，例如 AF_INET 為 sockaddr_in、AF_UNIX 是 sockaddr。參數 peeradrlen 是整數指標，在呼叫 accept() 時傳入 peeradr 的資料長度。Accept() 執行成功後，系統會回應對方 Socket 位址的實際長度，對 AF_INET 型態而言，該傳回值固定為 sizeof(sockaddr_in)，但對 AF_UNIX 而言，該傳回值並不固定（一般在 110 位元組以內）。

如果在非同步傳輸模式（Asynchronous），執行 accept() 之前沒有 Client 要求連線時，系統會回應 -1（newfd = -1），並告知錯誤訊息（errno）。在同步傳輸模式中，也會有其它因素造成執行錯誤，而回應 -1（newfd = -1），但可由錯誤訊息中瞭解執行失敗的原因。

8-5-5 利用 TCP 協定傳輸資料

利用 TCP 協定來傳輸資料，首先應使用連線程式（listen() 與 accept() 功能呼叫），待連線建立後，資料即固定在這條連線上流動，使用者就可以利用 read()/recv() 與 write()/send() 來傳輸資料。事實上，因為雙方連線已經建立，傳輸資料只須針對傳輸緩衝器作讀寫的動作，因此 write() 可以取代 send() 的動作，另外 read() 的動作也和 recv() 一樣。

(A) write()：將資料寫入 Socket 中，傳送給連線對方。

```
#include <sys/socket.h>
int sd;
int nbytes;
char *buf;
int ndata;

ndata = write(sd, buf, nbytes);
```

其實 write() 是將資料寫入傳送緩衝器上，再由下層 TCP/IP 通訊軟體負責傳送到對方，對方也是如此，下層 TCP/IP 通訊軟體負責將連線中的資料接收後，儲存於接收緩衝器，等待上層執行 read() 功能呼叫來讀取。write() 程式中的 sd 是執行 accept() 程式之後，所產生的 Socket 識別碼（newfd），參數 buf 是一個字元指標，指向欲寫入緩衝器的位址，nbyte 表示欲寫入多少個位元組資料，也就是傳送資料的長度。

當 write() 執行完後，會傳回確實所寫入的位元組數目 ndata，如果 ndata = -1 表示執行錯誤。這種現象表示呼叫 write() 時，所欲發送資料的數目並不一定一次可以發送完，也許必須經過多次的發送。這是因為執行 write() 時，系統只將資料佇列於傳送緩衝器上，再由底層負責傳送（TCP/IP 軟體），但如果寫入動作太快，而下層來不及傳送，便會發生緩衝器滿載的情形，此時系統無法接收新的資料。使用者必須檢查 ndata 所傳回的值，得知真正傳送出去的位元組。如果沒有全部傳送出去，可延遲一段時間後再傳送剩餘部分。

(B) read()：由連線中的 Socket 讀取資料

```
#include <sys/socket.h>
int sd;
int nbytes;
char *buf;
int ndata;

ndata = read(sd, buf, nbytes);
```

read() 的功能是由連線中的 Socket 讀取資料，即接收讀取緩衝器上的資料。其中 sd 是經過建立連線後（connect() 功能呼叫）所產生新的 Socket 識別碼，也就是連線的 Socket ID。buf 是一個字元指標，表示所接收資料的存放位址。nbytes 為預留接收位置的長度，表示此次執行 read() 最高可接收多少資料。當 read() 執行後，會回傳一個整數（ndata）表示確實接收到多少個位元組資料，如果回應 -1（ndata = -1）表示功能呼叫錯誤。

事實上，當建立連線後（connect()），下層 TCP/IP 通訊軟體已開始接收資料，而將收到的資料儲存於接收緩衝器上，等待應用程式執行 read() 來讀取資料。但如果緩衝器已滿，而上層還未讀取，則下層（TCP/IP）會通知對方暫緩傳送。

8-5-6 利用 UDP 協定傳輸資料

利用 UDP 協定傳輸資料時，不需要經過建立連線程式，只要雙方開啟 Socket 之後，便可利用 sendto() 和 recvfrom() 來傳送資料。資料在傳送當中必須註明對方 Socket 位址，接收時也必須由資料中判斷是由何地方傳送過來，因此在 sendto() 和 recvfrom() 兩個功能呼叫中的參數較為複雜。

(A) sendto()：將資料傳送給遠端 Socket

```
#include <sys/types.h>
#include <sys/socket.h>
int sd;
char *buf;
int len;
int flages;
struct sockaddr *tosd;
int tosdlen;
int ndata;

ndata = sendto(sd, buf, len, flages, tosd, tosdlen);
```

sendto() 是用來將資料傳送給對方，sd 表示本身 Socket ID（型態為 SOCK_DGRAM），buf 為欲傳送資料的指標位置，len 表示所要傳送資料的長度（位元組），參數 flags 是用來告知系統傳送的特性。參數 tosd 表示所要傳送對方 Socket 的位址，而 tosdlen 為對方 Socket 位址的長度，如以 AF_INET 為例，位址長度為 sizeof(struct sockaddr_in)。

和 read() 一樣，sendto() 並不保證所欲傳送的資料可以傳完，也許會有緩衝器滿載的現象，因此執行 sendto() 後，會傳回已成功傳送的位元組數目（ndata），如果傳回的值為 -1（ndata = -1）表示該功能呼叫失敗。

(B) recvfrom()：接收遠端 Socket 傳來的資料

```
#include <sys/types.h>
#include <sys/socket.h>


int sd;
char *buff;
int len;
int flags;
```

```
struct sockaddr *fromsock;
int *fsocklen;
int ndata;

ndata = recvfrom(sd, buff, len, flags, fromsock, fsocklen);
```

recvfrom() 是用來收取遠端 Socket 傳給本機資料的功能呼叫。其中參數 sd 表示本地 Socket 位址（經由 socket() 與 bind() 呼叫而成）；buff 為所接收資料存放位址的指標；len 為預留多少長度的緩衝器準備接收資料，以位元組計算；flags 是告知系統接收資料的工作模式。接收到對方傳送之資料時，也必須知道對方來自何處，呼叫者必須準備一個 Socket 資料結構的空間，來存放對方 Socket 的位址，此為 fromsock 參數，而對方位址的長度存放於 fsocklen 變數內，兩者皆為指標變數。

執行 recvfrom() 功能呼叫後，會傳回到底接收到幾個位元組資料（ndata），如果回應 -1（ndata = -1）表示執行錯誤。參數 flags 是用來告知系統傳送或接收資料的特性，其定義如下：

- **MSG_OOB：**接收或傳送 Out of Band 的資料，此一選項僅適用於 Stream Socket。
- **MSG_PEEK：**當設定此功能時，系統將資料傳送資料後，緩衝器還會保留原來資料。

8-5-7 管理 Socket 系統呼叫

在 Unix/Linux 上提供一些有關 Socket 管理的功能呼叫，分別敘述如下：

(A) getpeername()：獲取連線對方的 Socket 位址

```
#include <sys/socket.h>
int sd;
int *peeradrlen;
struct sockaddr *peeradr;
int status;

status = getpeername(sd, peeradr, peeradrlen);
```

getpeername() 是用來獲取連線對方 Socket 位址的功能呼叫。呼叫者必須提供一個空的 Socket 位址（peeradr）來接收該位址，而必須指定所承接緩衝器的長度（peeradrlen），兩者皆為位址指標。參數 sd 表示本身 Socket 的識別碼。如果執行正確會回應 0（status = 0）；否則回應 -1（status = -1）。

(B) getsockname()：取得本地 Socket 的位址

```
#include <sys/socket.h>
int sd;
int *myadrlen;
struct sockaddr *myadr;
int status;

status = getsockname(sd, myadr, myadrlen);
```

getsockname() 是用來讀取本地 Socket 位址的功能呼叫，其參數 myadr 及 myadrlen 如同 getpeername() 的定義相同。

(C) setsockopt()：設定 Socket 特性選項

```
#include <sys/socket.h>

int sd;
int level;
int optname;
int *optlen;
char *optval;
int status

status = getsockopt(sd, level, optname, optval, optlen);
```

setsockopt() 的功能是設定 Socket 的一些特性，這些特性會影響到 Socket 的運作情形。參數 sd 為欲設定之 Socket，而執行正常會回應 0 (status = 0)；否則回應 -1。參數 level 表示欲設定選項值是位於 TCP/IP 通訊協定之中哪一個介面，一般有三個主要設定介面：(如圖 8-6 所示)

(1) SOL_SOCKET：設定選項值位於 Socket 層次 (Level)。

(2) IPPROTO_TCP：針對 TCP 層次設定選項值，此設定方式只有對 TCP/IP 通訊模式有效。

(3) IPPROTO_IP：針對 IP 層次設定選項，也是僅對 TCP/IP 模式有效。

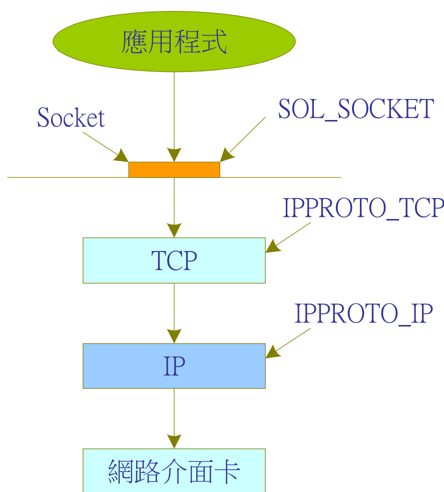

圖 8-6 Socket 設定選項層次

參數 optname、optval 與 opelen 表示設定選項的名稱，及其相對應之設定值和設定值的長度。有關選項值和設定層次有關，而且各家系統並不完全相同，使用時必須參考系統使用手冊才會正確，一般常用的設定選項有：

- **SO_BROADCAST：**設定通訊通道具有廣播的功能。
- **SO_DEBUG：**設定核心 (Kernel) 能結合通訊通道具有偵錯 (Debug) 資訊的功能。
- **SO_DONTROUTE：**設定資料能 Bypass 核心程式的路由選擇。
- **SO_KEEPALIVE：**設定虛擬電路能保持連線的功能。譬如在連線一端終止連接，而本通訊點還可保持連接狀態。
- **SO_LINGER：**設定如何處理當連線終止後，還保留在佇列器上等待傳送的資料。
- **SO_RCVBUF：**修改 Socket 的接收緩衝器空間大小。
- **SO_SNDBUF：**修改 Socket 的傳送緩衝器空間大小。

(D) getsockopt()：讀取 Socket 上的選項設定值

```
#include <sys/socket.h>


int sd, level, optname, *optlen;
char *optval;
int status;


status = getsockopt(sd, level, optname, optval, optlen);
```

getsockopt() 和 setsockopt() 的功能正好相反，getsockopt() 是用來取得目前 Socket 選項的設定值，參數表示方式也如同 setsockopt() 一樣。

(E) shutdown() 或 close()：關閉 Socket

```
int sd;
int status;


status = close(sd);
```

close() 將 Socket sd 關閉，以結束通訊連線。但可能還有訊息佇列在緩衝器（包括資料、確認訊號等等）尚未傳送出去。對於一般 TCP 協定而言，雖然執行 close() 後回即時回應完成訊息，其實 TCP/IP 通訊軟體還是會繼續將訊息傳送完畢，才真正完成斷線的工作。對於非連接方式（UDP）的 Socket 而言，理論上下層並不需要作斷線的動作，也就是說可以不用下 close() 功能呼叫，但是為了讓系統系放所佔有的資源，使用者還是必須下 close() 功能，以完成釋放的動作。

8-5-8 其它功能呼叫程式

以下介紹其它有關 Socket 程式發展的功能呼叫：

(A) 網路位元組次序和主機位元組次序轉換

```
unsigned long i, k;
k = htonl(i)    /* host to network byte order converter (long) */
```

```
unsigned short i, k;
k = htons(i)    /* host to network byte order converter (short) */


unsigned long i, k;
k = ntohl(i)    /* network to host byte order converter (long) */


unsigned short i, k;
k = ntohs(i)    /* network to host byte order converter (short) */
```

主機位元組表示的次序（Order）依照各種 CPU 型態有所不同，Motorola 和 Intel CPU 表示次序剛好相反，一般網路位址的表示次序和 Motorola CPU 相同，因此使用 Intel 系列的 CPU，其網路和主機位址的次序必須經過轉換才會相配合。htonl() 和 htons() 是將主機位元組次序轉換為網路位元組次序，一者為長整數（long）；而另一者為短整數（short）。ntohl() 和 ntohs() 是將網路位元組次序轉換成主機位元組次序。例如，i = 0x1234，則 htons(i) = 0x3412；若 i = 0x12345678，則 htonl(i) = 0x78563412。

(B) 區塊資料拷貝、清除或比較

```
char *src, *dest;
int len;
void bcopy(src, dest, len)


char *src;
int len;
void(src, len);


chat *src, *dest;
int len, status;
status = bcmp(src, dest, len);
```

在網路程式設計上常需要將資料由一個緩衝器複製到另一個緩衝器上（bcopy()）、或者清除緩衝器上的資料（bzero()）、或兩個緩衝器之間的內容做比較（bcmp()）。以 bcmp() 為範例，參數 src 表示來源資料所存放位址的指標，dest 表示被比較（目的）資料的指標，len 為比較資料的位元組長度，回應值 status = 0 表示兩個緩衝器的內容相同；否則為 status = -1。

8-6 虛擬電路程式範例

我們利用 Socket 程式集開發一個簡單的虛擬電路之程式範例，該程式是建立在 TCP 協定上。本程式範例是一種回應伺服器(Echo Server)，Client 端開啟某一檔案，將檔案內容傳送給 Server 端，Server 端收到訊息後再回送該訊息給 Client 端。本程式是在 RedHat Linux 上發展，相信一般 Unix 或 Linux 作業系統上，都可以馬上執行，而不需任何的修改。程式範例分為兩個部分：tcpsrv.c 與 tcpcln.c，一者為 Echo Server 程式；而另一者為 Echo Client 程式，程式編譯如下：

- **Server 程式編譯及執行：（Server 主機位址 163.15.2.30）**

```
# cc –o tcpsrv tcpsrv.c
# tcpsrv &
```

- Client 程式編譯及執行：（Client 主機位址 163.15.2.62）

```
# cc –o tcpcln tcpcln.c
# tcpcln 163.15.2.30 file_a
```

Client 端呼叫遠端伺服器（163.15.2.30），而將檔案 file_a 的內容傳送給伺服器，在伺服器的螢幕上會顯示該內容，伺服器再將所收到的資料回傳給客戶端，也會在客戶端的螢幕上顯示該內容。

8-6-1 TCP 伺服端程式範例

伺服器端程式 tcpsrv.c 的執行步驟如下：（請參考圖 8-4）

(1) 開啟 Socket（呼叫 socket()）。

(2) 建立位址格式（sockaddr_in）。

(3) 定址 Socket（呼叫 bind()）。

(4) 進入聆聽狀態（呼叫 listen()）。

(5) 等待連線要求（呼叫 accept()），如有連線要求，再進入下一步驟。

(6) 讀取 Client 端傳送的資料（呼叫 read()）。

(7) 顯示 Client 端傳送的資料。

(8) 回傳資料給 Client 端（呼叫 write()）。

(9) 回到步驟 5，等待下一個 Client 端要求連線。

程式範例如下：

```
/***   TCP Server（tcpsrv.c）
 *
 * 利用 socket 介面設計網路應用程式
 * 程式啟動後等待 client 端連線，連線後印出對方之 IP 位址
 * 並顯示對方所傳遞之訊息，並傳回給 Client 端。
 *
 */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/errno.h>

#define SERV_PORT 5134

#define MAXNAME 1024

extern int errno;

main()
{
    int socket_fd;      /* file description into transport */
    int recfd; /* file descriptor to accept           */
```

```
    int length;       /* length of address structure          */
    int nbytes;       /* the number of read **/
    char buf[BUFSIZ];
    struct sockaddr_in myaddr; /* address of this service */
    struct sockaddr_in client_addr; /* address of client       */
/*
 *          Get a socket into TCP/IP ***/
 */
    if ((socket_fd = socket(AF_INET, SOCK_STREAM, 0)) <0) {
        perror ("socket failed");
        exit(1);
    }
/*
 *      Set up our address
 */
    bzero ((char *)&myaddr, sizeof(myaddr)); /* 清除位址內容 */
    myaddr.sin_family = AF_INET;       /* 設定協定格式 */
    myaddr.sin_addr.s_addr = htonl(INADDR_ANY); /* IP 次序轉換 */
    myaddr.sin_port = htons(SERV_PORT);    /* 埠口位元次序轉換 */

/*
 *        Bind to the address to which the service will be offered
 */
    if (bind(socket_fd, (struct sockaddr *)&myaddr, sizeof(myaddr)) <0) {
        perror ("bind failed");
        exit(1);
    }
```

```
/*
 *   Set up the socket for listening, with a queue length of 20
 */
      if (listen(socket_fd, 20) <0) {
            perror ("listen failed");
            exit(1);
      }
/*
 *   Loop continuously, waiting for connection requests
 *   and performing the service
 */
      length = sizeof(client_addr);
      printf("Server is ready to receive !!\n");
      printf("Can strike Cntrl-c to stop Server >>\n");
      while (1) {
            if ((recfd = accept(socket_fd,
                  (struct sockaddr_in *)&client_addr, &length)) <0) {
                  perror ("could not accept call");
                  exit(1);
                  }

            if ((nbytes = read(recfd, &buf, BUFSIZ)) < 0) {
                  perror("read of data error nbytes !");
                  exit (1);
            }

            printf("Create socket #%d form %s : %d\n", recfd,
```

```
                inet_ntoa(client_addr.sin_addr), htons(client_addr.sin_port));
                printf("%s\n", &buf);

                /* return messages to client */
                if (write(recfd, &buf, nbytes) == -1) {
                        perror ("write to client error");
                        exit(1);
                }
                close(recfd);
                printf("Can Strike Crtl-c to stop Server >>\n");
        }
}
```

8-6-2 TCP 客戶端程式範例

客戶端程式為 tcpcln.c，它的執行步驟如下：(請參考圖 8-4)

(1) 檢查執行程式之參數。

(2) 開啟 Socket (呼叫 socket())。

(3) 建立位址格式。

(4) 定址 Socket (呼叫 bind())。

(5) 填入 Server 端位址。

(6) 連線至 Server 端 (呼叫 connect())。

(7) 開啟並讀取檔案。

(8) 傳送訊息給 Server 端 (呼叫 write())。

(9) 讀回並顯示訊息 (呼叫 read())。

程式範例如下：

```
/*    TCP    Client program (tcpclient.c)
 *
 *    本程式啟動後向 Server (tcpserver) 要求連線，
 *    並送出某檔案給 Server，再由 Server 收回該檔案
```

```
 *   並顯示該檔案內容於螢幕上
 *
 */

#include <stdio.h>
#include <netdb.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <fcntl.h>

#define SERV_PORT 5134
#define MAXDATA     1024

#define MAXNAME 1024
main(argc, argv)
int argc;
char **argv;
{
        int fd;                          /* fd into transport provider */
        int i;                           /* loops through user name */
        int length;                      /* length of message */
        int fdesc;                       /* file description */
        int ndata;                       /* the number of file data */
        char data[MAXDATA];             /* read data form file */
        char data1[MAXDATA];            /*server response a string */
        char buf[BUFSIZ];               /* holds message from server */
        struct hostent *hp;             /* holds IP address of server */
        struct sockaddr_in myaddr;      /* address that client uses */
        struct sockaddr_in servaddr;    /* the server's full addr */

        /*
         * Check for proper usage.
         */
        if (argc < 3) {
                fprintf (stderr,
                        "Usage: %s host_name(IP address) file_name\n", argv[0]);
                exit(2);
```

```
}
/*
 *   Get a socket into TCP/IP
 */
if ((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror ("socket failed!");
        exit(1);
}
/*
 * Bind to an arbitrary return address.
 */
bzero((char *)&myaddr, sizeof(myaddr));    /* 清除位址內容 */
myaddr.sin_family = AF_INET;               /* 設定協定格式 */
myaddr.sin_addr.s_addr = htonl(INADDR_ANY); /* IP 次序轉換 */
myaddr.sin_port = htons(0);          /* 埠口位址任意 */

if (bind(fd, (struct sockaddr *)&myaddr,
                sizeof(myaddr)) <0) {
        perror("bind failed!");
        exit(1);
}
/*
 * Fill in the server's address and the data.
 */

bzero((char *)&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(SERV_PORT);    /* 設定 Server 埠口 */

hp = gethostbyname(argv[1]);
if (hp == 0) {
        fprintf(stderr,
                "could not obtain address of %s\n", argv[2]);
        return (-1);
}

bcopy(hp->h_addr_list[0], (caddr_t)&servaddr.sin_addr,
```

```
        hp->h_length);                /* 設定 Server 的 IP 位址 */
    /*
     * Connect to the server 連線.
     */
    if (connect(fd, (struct sockaddr *)&servaddr,
            sizeof(servaddr)) < 0) {
        perror("connect failed!");
        exit(1);
    }
    /**開起檔案讀取文字 **/
    fdesc = open(argv[2], O_RDONLY);
    if (fdesc == -1) {
        perror("open file error!");
        exit (1);
    }
    ndata = read (fdesc, data, MAXDATA);
    if (ndata < 0) {
        perror("read file error !");
        exit (1);
    }
    data[ndata] = '\0';

    /* 發送資料給 Server */
    if (write(fd, data, ndata) == -1) {
        perror("write to server error !");
        exit(1);
    }
    /** 由伺服器接收回應 **/
    if (read(fd, data1, MAXDATA) == -1) {
        perror ("read from server error !");
        exit (1);
    }
    /* 印出 server 回應 **/
    printf("%s\n", data1);

    close (fd);
}
```

8-7 電報傳輸程式範例

如同虛擬電路之範例程式一樣功能，我們再開發電報傳輸模式而建構在 UDP 協定上。Client 端和 Server 端範例程式為：updsrv.c 與 udpcln.c，編譯與執行方式也和虛擬電路範例一樣：

- **Server 程式編譯及執行：(Server 主機位於：163.15.2.30)**

  # cc –o udpsrv udpsrv.c

  # udpsrv &

- **Client 程式編譯及執行：(Client 主機位於：163.15.2.62)**

  # cc –o udpcln udpcln.c

  # udpcln 163.15.2.30 file_a

8-7-1 UDP 伺服端程式範例

UDP 伺服端程式為 udpsrv.c，程式範例如下：(請參考圖 8-5，執行步驟大略和 TCP 相同，不再另述)

```
/***    UDP Server(udpsrv.c)
 *
 * 利用 socket 介面設計網路應用程式
 * 程式啟動後等待 client 端連線，連線後印出對方之 IP 位址
 * 並顯示對方所傳遞之訊息，並回送給 Client 端。
 *
 */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/errno.h>

#define SERV_PORT 5134

#define MAXNAME 1024
```

```
extern int errno;

main()
{
        int socket_fd;      /* file description into transport */
        int recfd;  /* file descriptor to accept                */
        int length;        /* length of address structure          */
        int nbytes;        /* the number of read **/
        char buf[BUFSIZ];
        struct sockaddr_in myaddr; /* address of this service */
        struct sockaddr_in client_addr; /* address of client        */
/*
 *          Get a socket into UDP/IP
 */
        if ((socket_fd = socket(AF_INET, SOCK_DGRAM, 0)) <0) {
                perror ("socket failed");
                exit(1);
        }
/*
 *      Set up our address
 */
        bzero ((char *)&myaddr, sizeof(myaddr));
        myaddr.sin_family = AF_INET;
        myaddr.sin_addr.s_addr = htonl(INADDR_ANY);
        myaddr.sin_port = htons(SERV_PORT);

/*
 *        Bind to the address to which the service will be offered
 */
        if (bind(socket_fd, (struct sockaddr *)&myaddr, sizeof(myaddr)) <0) {
                perror ("bind failed\n");
                exit(1);
        }

/*
 *   Loop continuously, waiting for datagrams
 *   and response a message
 */
        length = sizeof(client_addr);
```

```
        printf("Server is ready to receive !!\n");
        printf("Can strike Cntrl-c to stop Server >>\n");

        while (1) {
                if ((nbytes = recvfrom(socket_fd, &buf, MAXNAME, 0,
                                &client_addr, &length)) <0) {
                        perror ("could not read datagram!!");
                        continue;
                        }


                printf("Received data form %s : %d\n",
                inet_ntoa(client_addr.sin_addr), htons(client_addr.sin_port));
                printf("%s\n", &buf);

                /* return to client */
                if (sendto(socket_fd, &buf, nbytes, 0, &client_addr, length)
                        < 0) {
                        perror("Could not send datagram!!\n");
                        continue;
                }
                printf("Can Strike Crtl-c to stop Server >>\n");
        }
}
```

8-7-2 UDP 客戶端程式範例

UDP 客戶端程式為 tcpcln.c，程式執行步驟大略和 TCP 客戶端相同不再另述，程式範例如下：（請參考圖 8-5）

```
/*   TCP   Client program (tcpcln.c)
 *
 *   本程式啟動後向 Server (tcpserver) 要求連線，
 *   並送出某檔案給 Server，再由 Server 收回該檔案
 *   並顯示出該檔案的內容
 *
 */
```

```
#include <stdio.h>
#include <netdb.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <fcntl.h>

#define SERV_PORT 5134
#define MAXDATA      1024

#define MAXNAME 1024
main(argc, argv)
int argc;
char **argv;
{
        int fd;                              /* fd into transport provider */
        int i;                               /* loops through user name */
        int length;                          /* length of message */
        int size;                            /* the length of servaddr */
        int fdesc;                           /* file description */
        int ndata;                           /* the number of file data */
        char data[MAXDATA];                  /* read data form file */
        char data1[MAXDATA];                 /*server response a string */
        char buf[BUFSIZ];                    /* holds message from server */
        struct hostent *hp;                  /* holds IP address of server */
        struct sockaddr_in myaddr;           /* address that client uses */
        struct sockaddr_in servaddr;         /* the server's full addr */

        /*
          * Check for proper usage.
          */
        if (argc < 3) {
                fprintf (stderr,
                        "Usage: %s host_name(IP address) file_name\n", argv[0]);
                exit(2);
        }
        /*
          *   Get a socket into TCP/IP
          */
```

```
if ((fd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror ("socket failed!");
        exit(1);
}
/*
  * Bind to an arbitrary return address.
  */
bzero((char *)&myaddr, sizeof(myaddr));
myaddr.sin_family = AF_INET;
myaddr.sin_addr.s_addr = htonl(INADDR_ANY);
myaddr.sin_port = htons(0);

if (bind(fd, (struct sockaddr *)&myaddr,
                sizeof(myaddr)) <0) {
        perror("bind failed!");
        exit(1);
}
/*
  * Fill in the server's UDP/IP address
  */

bzero((char *)&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(SERV_PORT);
hp = gethostbyname(argv[1]);

if (hp == 0) {
        fprintf(stderr,
                "could not obtain address of %s\n", argv[2]);
        return (-1);
}

bcopy(hp->h_addr_list[0], (caddr_t)&servaddr.sin_addr,
        hp->h_length);

/**開起檔案讀取文字 **/

fdesc = open(argv[2], O_RDONLY);
if (fdesc == -1) {
        perror("open file error!");
```

```
            exit (1);
        }
        ndata = read (fdesc, data, MAXDATA);
        if (ndata < 0) {
            perror("read file error !");
            exit (1);
        }
        data[ndata + 1] = '\0';

        /* 發送資料給 Server */
        size = sizeof(servaddr);
        if (sendto(fd, data, ndata, 0,
            &servaddr, size) == -1) {
            perror("write to server error !");
            exit(1);
        }
        /** 由伺服器接收回應 **/
        if (recvfrom(fd, data1, MAXDATA, 0, &servaddr, &size) < 0) {
            perror ("read from server error !");
            exit (1);
        }
        /* 印出 server 回應 **/
        printf("%s\n", data1);

}
```

8-8 Socket 多工方式

Socket 多工方式是表示一個 Socket 可否同時接受遠端多個通訊端點的連線要求，或是一個應用程式同時可以監視多個 Socket 通訊端點的連線。基本上，利用非同步傳輸模式的通訊鏈路，就可以達到此目的。但非同步傳輸的訊號（Signal）處理必須透過系統中斷程式，來告知使用者已發生事件（資料進入），如此牽涉到中斷系統，導致使用者編寫應用程式上較為困難。因此，我們可以在不變更系統環境之下，採用下列兩種方法達到 Socket 多工的目的：

當 Socket 被執行 listen() 系統呼叫（虛擬電路方式），而進入聆聽狀態之後，不要即時執行 read() 呼叫（或 recvfrom() 呼叫）。而先利用 select() 去詢問連線是否有資料進來，如果資料已進入，在執行 read() 呼叫來讀取該資料，如此執行 read() 呼叫就不會無窮的等待著。因此，一個主程式就可以監視多個 Socket 端點要求連線，或處理多個端點的傳輸資料。

另一種方法是 Socket 隨時監視是否有訊號進入，也就是直接執行 read() 系統呼叫等待接收訊號。但當收到遠端連接訊號後，立即利用 fork() 系統呼叫產生子程序（Child Process），由子程序負責連線處理的工作，而原來主程序（Main Process）再回去監視是否有其它遠端要求連線，如此，便可以達到一個 Socket 的通訊端點，可以接受多個使用者要求連線。一般 Internet 上的 Client/Server 架構上的伺服器都可同時接受多個使用者連線，下一節在詳加介紹其製作方法。

如將上述兩種方法可以混合使用，便可以達到 Socket 多工的主要功能：『一個主程式可以監視或處理多個 Socket 端點連線，並且一個 Socket 端點可以接受多個使用者的連線』，以下分別介紹這兩種多工方式：

(1) 當 Socket 被執行 listen() 系統呼叫（虛擬電路方式），而進入聆聽狀態之後，不要即時執行 read() 呼叫（或 recvfrom() 呼叫）。而先利用 select() 去詢問連線是否有資料進來，如果資料已進入，在執行 read() 呼叫來讀取該資料，如此執行 read() 呼叫就不會無窮的等待著。因此，一個主程式就可以監視多個 Socket 端點要求連線，或處理多個端點的傳輸資料。

(2) 另一種方法是 Socket 隨時監視是否有訊號進入，也就是直接執行 read() 系統呼叫等待接收訊號。但當收到遠端連接訊號後，立即利用 fork() 系統呼叫產生子程序(Child Process)，由子程序負責連線處理的工作，而原來主程序（Main Process）再回去監視是否有其它遠端要求連線，如此，便可以達到一個 Socket 的通訊端點，可以接受多個使用者要求連線。一般 Internet 上的 Client/Server 架構上的伺服器都可同時接受多個使用者連線，下一節在詳加介紹其製作方法。

如將上述兩種方法可以混合使用，便可以達到 Socket 多工的主要功能：**『一個主程式可以監視或處理多個 Socket 端點連線，並且一個 Socket 端點可以接受多個使用者的連線』**，以下分別介紹這兩種多工方式：

8-8-1 Socket 多工輸入/輸出

Socket **『多工輸入/輸出』（Multiple I/O）**是表示一個主程式可允許多個 Socket 端資料的進出。早期 Unix 系統大多利用 poll() 系統呼叫來監視多個檔案識別碼，首先將一串列的檔案識別碼加入 poll() 函數內。執行 poll() 後，在這些檔案識別碼之中有任一個事件發生，poll() 函數就會回應給主程式。目前大部分系統都採用 select() 系統呼叫來取代 poll()，而檔案識別碼也相當於建構 Socket 所產生的 Socket ID，它的工作模式如圖 8-7 所示。簡單的說，雖然每一應用程式都銜接到各自的 Socket 埠口上，但利用一個監督程式來監視若干個 Socket 埠口是否有連線要求，而並非每一應用都各自監督埠口。我們的做法是用一個集合變數 fd_set 來存放被監視的 Socket 端點的識別碼：fd_set = (socket_1, socket_2, socket_3, socket_4, socket_5)，而其中任何一個 Socket 有事件發生（Read, Write, Exception），便會回應給主程式，再由主程式來處理該事件。

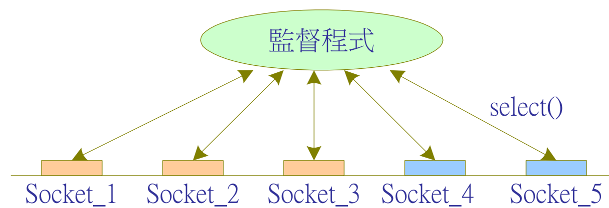

圖 8-7 Socket 多工讀寫的運作

select() 系統呼叫的格式如下：

```
#include <sys/types.h>
#include <sys/time.h>

struct timeval *timeout;
int            numfds;
fd_set         *rfds;
fd_set         *wfds;
fd_set           *efds;
int              numsel;

numsel = select(numfds, rfds, wfds, efds, timeout);
```

各參數功能如下：

- **rfds**：此參數的作用在告訴系統有哪些 Socket 通訊端點準備作讀取的動作(Ready to Read)。參數 rfd 是以 fd_set 資料結構存放，使用者將已開啟而準備接收連線或資料的 Socket，存放在 rfd 上以執行 select() 來告訴系統。譬如 rfd = {32, 45, 12}，表示準備接收 Socket 32、45 和 12 的連線訊號或資料，這也是監視這幾個傳輸埠口的意思。當系統監視這幾個埠口有訊號進來，還是以 rfd 參數回應給使用者，使用者由 rfd 參數中知道有哪幾個 Socket 已有訊號進來，而必須去讀取 (read())。

- **wfds**：告訴系統有哪幾個 Socket 端點準備傳送資料 (Ready to Write)，參數 wfd 也是 fd_set 的資料結構。譬如 wfd = {24, 12, 45}，表示 Socket 24、12 和 45 準備傳送資料。使用者利用 select() 系統呼叫告訴系統，系統就處理一些下層連線的動作。如果下層連線完成，便可以傳送資料時，但系統也是利用 wfd 參數回應給使用者，使用者便可以由回應的 wfd 中得知哪幾個 Socket 已可以發送資料，如此就不會被**『阻斷』(Blocking)**。

- **efds**：功能如同 efds 和 rfds，但它是針對異常事件 (Exception Event) 的監視與回應。

- **timeout**：適用於設定執行 select() 等待的時間值。當執行 select() 呼叫時，有時候並無法即時回應所監視 Socket 的反應，timeout 表示可以最長的時間，簡單的說，表示 select() 被 Blocking 的時間。timeout 是資料結構 timeval 的指標，其結構如下：

```
struct timeval {
        long        tv_sec;     /* seconds          */
        long        tv_usec     /* microseconds     */
};
```

其中 tv_sec 表示等待的秒數；而 tv_usec 為微秒數。

- **numfds**：表示此次執行 select() 中最大的 Socket 的識別值 (Socket ID) 加一。

- **numsel**：執行 select() 的回應值，表示發生幾個事件的總和，也就是 rfds、wfds 和 efds 內 Socket ID 數目的總和。如 numsel = −1 表示執行錯誤。

select() 系統呼叫中的 timeout 值有下列三種執行方式：

- **timeout 是空值 (Null) 指標**：select() 呼叫會無窮盡的等待，直到 rfds、wfds 或 efds 中有任何一個事件發生，才會返回。

- **timeout 不是空值指標，但 tv_sec 和 tv_usec 皆為 0：**select() 會檢查所指定的 Socket 一次，不論是否有事件發生，都會立即返回。
- **timeout 不是空值指標，且 tv_sec 和 tv_used 都有值：**當 select() 檢查所指定的 Socket 後，如有事件發生會及時返回，否則會等待 Socket 的事件，直到所指定時間完了，才會返回。

在執行 select() 後返回的 rfds、wfds 與 efds 內所儲存的 Socket ID，表示這些 Socket 都必須進一步的處理。執行前後這三個指標變數的內容含義不同，執行前表示必需偵測的 Socket ID；執行後表示有發生事件的 Socket ID，因此，執行程式之前必須將原有的指標內容保存，才不會被覆蓋。但 rfds、wfds 和 efds 都是 fd_set 的指標變數，在包含檔 sys/types.h 中描述。fd_set 的內容也是整數變數，利用整數中每一位元值表示相對應的 Socket ID，如果某一位元被設定為 1，則表示該 Socket 被設定。。另外，為了讓使用者方便設定和測試 rfds、wfds 和 efds，Socket Library 提供一系列的巨集函數來處理，以減低系統發展的複雜性，巨集函數如下：

```
#include <sys/types.h>
int fd;                    /* Socket ID   */
fd_set *fdset;             /* fdset 為 fd_set 的指標變數 */
FD_ZERO(fdset);            /* 將 fdest 內所有位元清除為 0 */
FD_SET(fd, fdset);         /* 將 fdset 中的 fd 位元設定為 1*/
FD_CLR(fd, fdset);         /* 將 fdset 的 fd 位元清除為 0 */
FD_ISSET(fd, fdset);       /* 測試 fdset 中 fd 位元是否被設定為 1 */
```

為了說明這些巨集函數的使用方法，假設已有準備讀取的 Socket 為 {2, 5, 7}、寫入為 {0, 2, 5}、異常事件為 {5, 8}，其設定如下：

```
fd_set *rfds, *wfds;*efds;
FD_ZERO(rfds); FD_ZERO(wfds); FD_ZERO(efds);    /* clear to fd_set*/
FD_SET(2, rfds); FD_SET(5, rfds); FD_SET(7, rfds);    /* ready to read */
FD_SET(0, wfds); FD_SET(2, wfds); FD_SET(5, wfds); /* ready to write */
FD_SET(5, efds); FD_SET(8, efds);                  /* ready to exception */
```

8-8-2 Socket 多工連線

『多工連線』(Multiple Connection) 表示一個 Socket 通訊端點可同時接受多個連線，一般 Client/Server 架構的伺服器所提供的服務，就必須具有此功能，才能服務多個使用者。Socket 的多工連線的運作方式如圖 8-8 所示。圖中伺服器端有一個應用程式，隨時聆聽 Socket (163.15.2.30:80) 是否有連線要求。當客戶端 (163.15.2.45:3452) 有連線過來時，伺服器端接收到連線要求，便以 fork() 系統呼叫產生一個子程式 (A 程式)，並將系統環境複製 (dup()) 到子程式，再由子程式負責和客戶端通訊。監督程式便關閉自己的連線端點，並回到繼續監督的狀況下，繼續等待是否有新的連線要求。

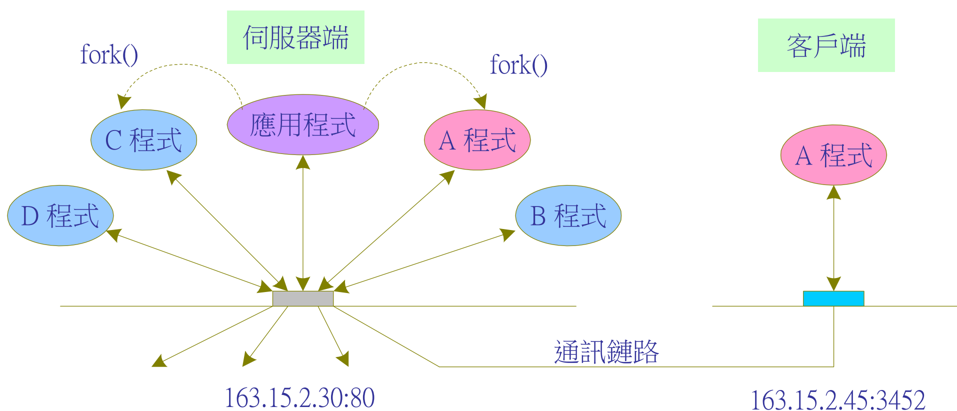

圖 8-8 Socket 多工連線的運作方式

圖 8-9 為虛擬電路 (TCP 連線) 的多工連線範例。但在這種情況之下，監督程式只能監視一個 Socket 通訊端點，一般都應用於使用較頻繁的伺服軟體上，譬如，網頁伺服器或郵件伺服器等等。但在一般網路上有許多使用率較低的伺服軟體，也必須隨時監視連線的要求，如果每個伺服器都自行監視，那會增加系統的複雜性，因此為了簡化系統的管理，可將多個 Socket 由一個監督程式來監視即可，但它接收到連線時，再產生子程式來處理它的連線動作，如此就能達成 Socket 的多工連線的功能。其實要達到此目的也不是困難，只要將圖 8-4 中 Server 端加入 fork() 和 dup() 功能呼叫即可，加入後具有多工連線的功能如圖 8-9 所示，我們在下一節將以 xinetd 為範例來說明製作過程。

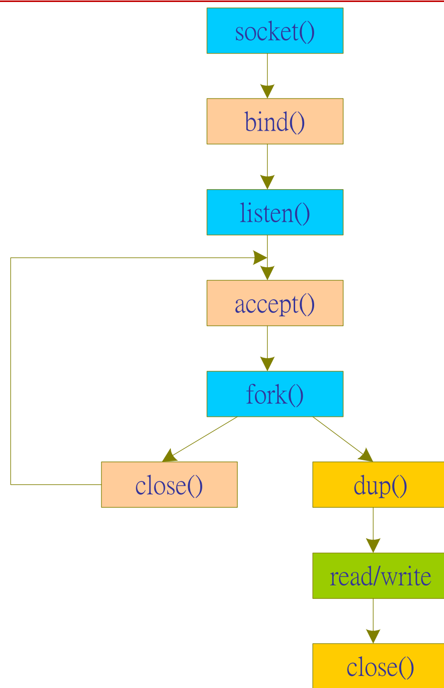

圖 8-9 Socket 多工連線的功能圖

8-8-3 Socket 多工範例 – xinetd

一般在 Unix/Linux 系統下隨時有許多服務程式（Daemon）在執行，譬如，Telnet、FTP、TFTP 等等，如果這些服務程式都有專門的監督程式在準備接受連線的話，那整個系統環境有就便得非常複雜，而且在管理上也較困難。因此我們可以將一些較少使用或較需要特殊處理的服務程式，由一個專門的監督程式來管理，這就是 inetd/xinetd 服務程式。有關 inetd 或 xinetd 的功能及設定已在本書第九章（9-9 節）中介紹，這裡只針對 xinetd 的工作原理加以解說。

(A) xinetd 的多工原理

xientd 守護程式是真正符合 Socket 的多工要求：**『一個主程式可以監視或處理多個 Socket 端點連線，並且每一個 Socket 端點可以接受多個使用者的連線』**。也就是說，xientd 符合**『多工輸入出』**和**『多工連線』**的要求，我們用圖 8-10 來說明它的多工原理。xientd 守護程式同時監督若干個應用程式（如 /etc/xinetd.d 目錄下），每一應用程式銜接到各自的傳輸埠口（如 /etc/services 檔案描述），譬如，telnet 連接到 23/tcp、tftp 連接到 69/udp 等等。當 xientd 監視到某一傳輸埠口（如 23/tcp）有連線要求時（圖中訊號 1），便呼叫該埠口的服務程式（如 in.telnet 服務程式），並將所有連線處理工作轉交給服務程式，再由服務程式（如 in.telnetd）直接和客戶端通訊（如圖中訊號 2）。xinetd 便釋放掉該處理動作，再回到監視埠口的工作，如此，xientd 便可達到多工輸入/輸出和多工連線的功能，我們也可以發現圖 8-10 是圖 8-7 和 8-8 的結合體。

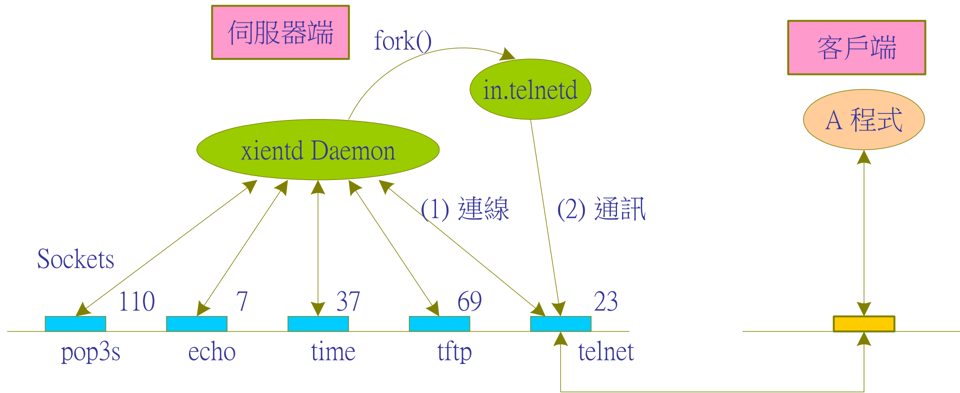

圖 8-10 xinetd 的多工原理

(B) xinetd 的運作程序

當系統啟動時（/etc/rc.d），xinetd 呼叫 /etc/xinted.d 下所有的服務程式，如果程式設定檔中 disable = no，表示該服務程式會被啟動，運作程序如圖 8-11 所示，說明如下：

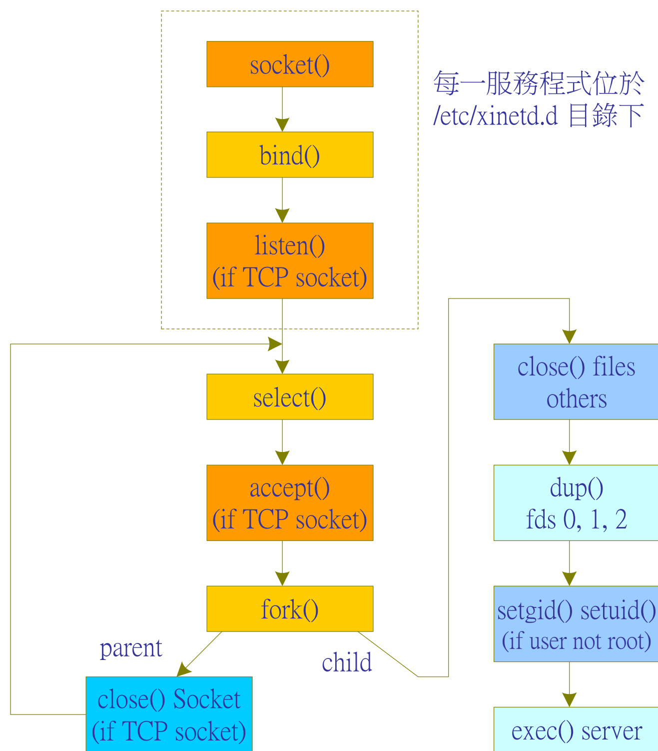

圖 8-11 xinetd 的運作程序

(1) 由 /etc/rc.d/init.d 啟動 xinetd 服務程式。

(2) xinetd 服務程式呼叫啟動 /etc/xinetd.d 目錄下的服務程式（如 telnet）。如果該程式設定檔有設定啟動（disable = no），就依照 /etc/services 內所設定之服務程式的傳輸埠口。以下是 /etc/services 的部分檔案內容，可以看出 Telnet 服務的傳輸埠口為 23，而可採用 TCP

或 UDP 傳輸。每一個服務程式呼叫的程序就如圖 8-11 中虛線所包圍的部分，如果在設定內規劃為 TCP 連線（socket_type = stream），才需要 listen() 系統呼叫。

```
ftp-data    20/tcp
ftp-data    20/udp
ftp         21/tcp
ftp         21/udp
ssh         22/tcp                  # SSH Remote Login Protocol
ssh         22/udp                  # SSH Remote Login Protocol
telnet      23/tcp
telnet      23/udp
```

(3) xinetd 將各個服務程式所產生的檔案描述子（File Descriptor）填入 select() 函數的 rfds 變數內，並執行 select() 監督所有開啟的 Socket 埠口。

(4) 當 select() 接收到遠端連線要求，便執行 accept() 接受連線，而得到一個連線的檔案描述子（表示遠端連線埠口）。此時 xinetd 以 fork() 呼叫產生一子程序，而由子程序負責連線的處理。子程序首先關閉除了此連線外，其它所有的檔案描述子，並以連線的檔案描述子呼叫 dup2() 產生檔案描述子 0、1 和 2，再關閉原來連線的檔案描述子。如果設定檔中所表示此程序所有人不是系統管理者（user = root），便再呼叫 setgid() 和 setuid() 改變程序所有人。其它還有許多關於安全性的設定方式，我們不再另外敘述請參考 xinetd 的使用手冊。

(5) 子程式以 exec() 呼叫該連線的服務程式，譬如 Telnet 服務程式為 /usr/sbin/in.telnetd。

(6) 如果是 TCP 連線（stream），則關閉原來連線（由子程式取代了），再繼續監督所有埠口（執行 select()）；否則直接再監督埠口是否有其連線進來。

在服務程式的設定檔（如 /etc/xinetd.d/telnet）內設定參數 wait，是這表示主程式 xinetd 是否等待子程式執行完畢後再回到監督狀態下。一般 TCP 連線都將其設定在沒有等待（wait = no），表示呼叫子程式後，主程式直接回到監督狀態，這是因為連線已建立完成，爾後子程式只要依照連線傳輸資料就不會遺失。但如果使用 UDP 協定，它並沒有建立連線，在當主程式接收到第一筆資料以後，並不能保證還有沒有下一筆同一來源的資料到達。如果主程式直接回到監督狀態，他收到

下一筆資料會再重新 fork() 一個子程式，造成重複的連線狀態，因此在 UDP 協定大多設定為有等待（wait = yes）。設定為有等待，有時候會因為每一連線的不正常，造成整個 xinetd 處理效率會不幅度降低。為了克服這種問題，目前在 Internet 網路上以 UDP 協定傳送訊息，儘可能限制在一個封包以內（512 Bytes），便可減少許多困擾。

習題

1. 何謂**『插座』（Socket）**？

2. 請說明 Socket 的連線方式。

3. 請說明 Socket 的六個基本功能。

4. 請依照 Client/Server 模式，繪圖說明 Socket 的虛擬電路連接方式。

5. 何謂**『阻斷模式』（Blocking Mode）**？何謂**『非阻斷模式』（Non-blocking Mode）**？兩者在 Socket 傳輸模式中有何不同？

6. 請依照 Clinet/Server 模式，繪圖說明 Socket 的電報傳輸連接方式。

7. 請依照 RFC 954 定義，利用 Socket 系統呼叫製作一個 whois 伺服器，讓客戶端查詢伺服器上登錄使用者的訊息，並由客戶端顯示出伺服器主機名稱與已登入使用者名稱。

8. 請利用 Socket 系統呼叫製作一個 Echo 伺服器，客戶端連線成功後回應客戶端的 IP 與傳輸埠口位址。

9. 何謂 Socket 的**『多工輸入/輸出』（Multiple I/O）**？請說明其運作原理。

10. 何謂 Socket 的**『多工連線』（Multiple Connection）**？請說明其工作原理。

11. 請簡述 xinetd 的多工技術。

12. 請製作一個超級守護程式，可以監督 whois 伺服器（第七題）和 echo 伺服器（第八題）的連線要求，並可呼叫客戶端要求的連線服務。