Introduction to UNIX Socket

簡介

- 網際網路提供基本的通訊服務，通訊協定軟體無法主動和遠端電腦通訊，而是由兩端的應用程式主導整個通訊：一端啟動通訊，而另一端接受之。
- 應用程式並不一直等待任何訊息到達，而是期待在和外界通訊前先和通訊協定軟體溝通。應用程式告知本機的通訊軟體，其所期待的特定訊息型態為何。當碰巧有吻合的新進訊息出現時，通訊協定軟體會將訊息轉交給等待的應用程式。

簡介 (Cont.)

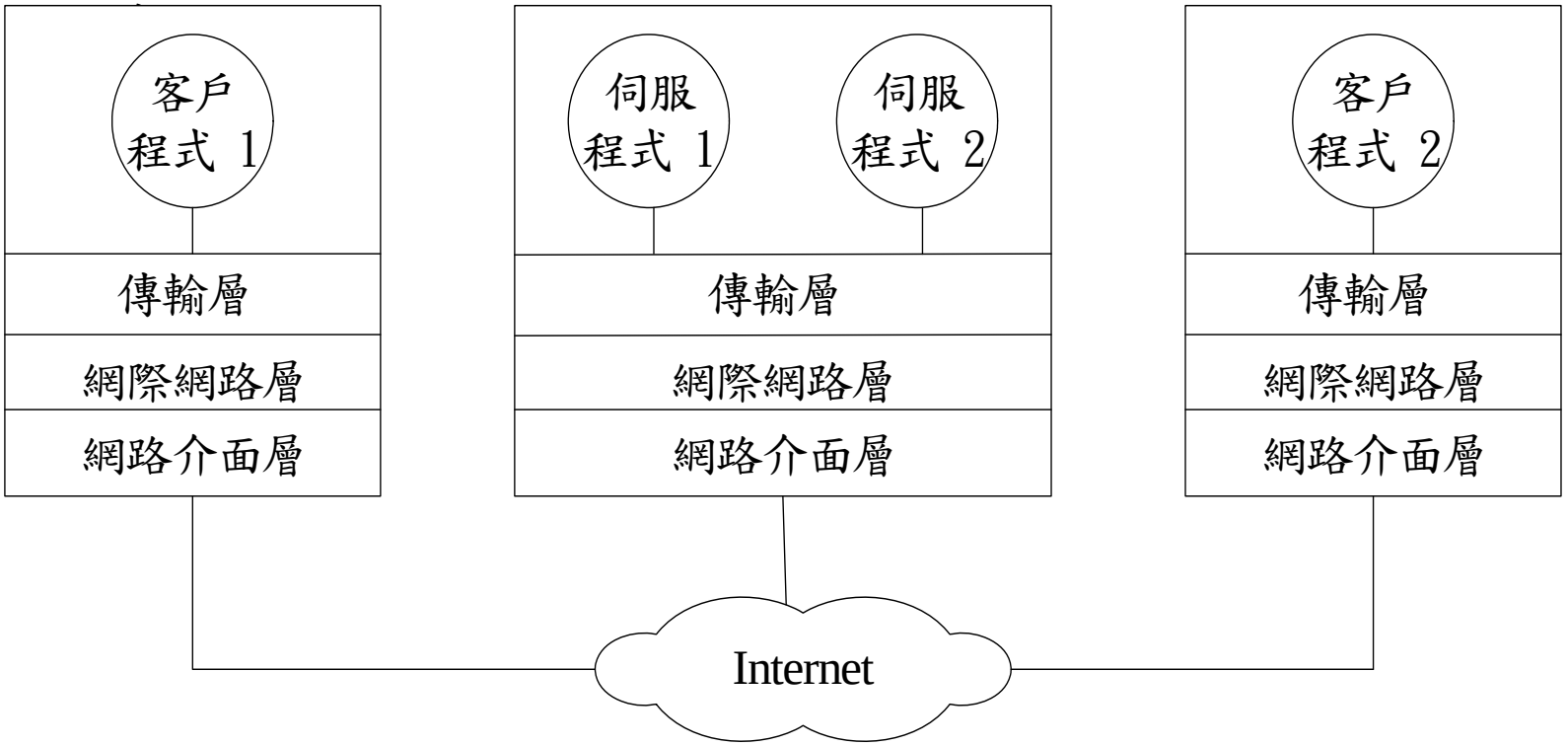
- 當然要通訊的兩個應用程式不能同時等待相同的訊息型態，必須是一方主動開始，此時另一方被動的等待。
- 如此互動的方式為主從 (client-server) 架構的互動。主動啟始聯繫的**應用程式**稱為**客戶端 (client)**，被動等待聯繫的程式稱為**伺服器端 (server)**。

簡介 (Cont.)

- Client 與 server 之間可以兩方向互送資訊。典型的例子是， client 送出一個要求給 server，而 server 送出一個回應給 client。
  - Client 送出一連串的要求給 server，而 server 也回送一連串回應給 client。
  - Server 提供連續輸出而不用 client 要求，只要 client 一連上 server，server 便開始傳送資料的作業。
  - Server 也可以接收新資訊與傳送新資訊 (Ex: file server)。

在一部電腦上提供多種服務

- 允許一部電腦執行多個伺服程式，可以共享此部電腦的硬體資源，減輕管理者的負荷。並可提高運用效



特定服務的識別

- 傳輸通訊協定提供一個允許客戶程式能清楚的指定其對應的伺服程式的機制。
  - 此機制即是分配給每種服務唯一的識別碼(identifier)。當伺服器程式開始執行時，它必須將代表其所提供服務的識別碼，註冊至本機的通訊協定的軟體裏。
  - 當 client 送出一個要求時，client 的傳輸通訊協定傳送此識別碼至 server 的傳輸通訊協定，然後 server 的傳輸通訊協定再根據此識別碼來選擇由哪一個伺服程式來處理 client 端的請求。

特定服務的識別 (Cont.)

- TCP 用一16 bits 正整數的服務識別碼稱之為通訊協定埠號碼 (protocol port number)，對應到某一個服務。
- 伺服程式指定代表其所提供的通訊協定埠號碼後，即進入被動等待通訊的狀態。
  - 在 server 端的 TCP 軟體程式根據新進訊息的通訊協定埠號碼來決定那個伺服程式必須接收這個要求。

單一伺服服務的多重複製

- 電腦系統中允許多個程式在同一時間執行的功能稱為 concurrency (同時執行)。
- 一個程式可以有不只一個執行緒 (thread) 執行。
- 因此一個具有 concurrency 功能的伺服程式可提供多個 client 同時存取，而不需要某一個 client 等待另一 client 程式執行完畢。
  - 當接到某一 client 要求時，伺服程式將要求交給一個新的 thread 執行處理，而這個 thread 和其他已經存在的 thread 同時繼續執行。

動態伺服程式的建立

- 大多數的伺服程式都是動態運作的，一收到新要求就相對建立一個新的 thread。實際上，伺服程式可分為兩個部分
  - 一為接受要求與建立新的 thread 的程式碼
  - 另一為處理個別要求的程式碼
- 當 concurrent 伺服程式開始執行時，只有第一部分會被執行。也就是說伺服程式的主執行緒是用來等待新的請求。當要求到達時，主執行緒會建立一個新的服務執行緒來處理此要求。此服務執行緒處理完要求後，即可終止。但主執行緒仍將一直維持伺服程式在執行的狀態，繼續等待下一個新的要求。

動態伺服程式的建立 (Cont.)

- 因此若有 N 個 client 存取單一電腦上的特定服務，則整個服務將有 N+1 個 thread。

傳輸通訊協定和不矛盾通訊

- 以 TCP 為例，client 端需選擇一本機尚未使用的 port number ，當它送出一 TCP segment 時，client 端程式需將此 port number 填入 TCP segment 的來源埠 (source port) 欄位，並將伺服程式的 port number 填入目的地埠 (destination port) 欄位。
- TCP 將合併來源埠與目的地埠號碼，作為識別某一特定連線時使用。因此所有的訊息得以從多個 client 端程式使用相同的服務，而不會分佈清楚這是哪一個 client 的訊息。

透過多種通訊協定的服務通達性

- 伺服程式不一定要在 connection oriented 和 connectionless 傳輸方式上做選擇，它也可以同時提供兩種以上的傳輸通訊協定來存取相同的服務，而把選擇權留給 client 端程式決定。
- 有兩種設計多種通訊協定服務的實做方法
  - 直接的方法：用兩個提供相同服務的伺服程式，一個用 connection oriented，另一個用 connectionless 傳輸。
  - 第二個方法用單一個伺服程式同時和兩種以上的傳輸通訊協定互動。此伺服程式接收各種通訊協定送來的要求，以其通訊協定來送回應。

複雜的主從架構互動

- 不限制一個客戶程式存取單一種服務：應用程式能首先扮演某一服務的客戶程式，接著在當令一服務的客戶程式。此客戶程式和不同的伺服程式 (有可能在不同的電腦上) 存取不同的服務。
- 不限制一個客戶程式對單一個伺服程式存取單一種服務：有些服務是在不同的電腦上的伺服程式會提供不一樣的資訊。例如：時間伺服程式提供所在電腦的現在時刻與日期資訊。在不同時區的伺服器會給不同的答案。而有些服務則會提供完全相同的答案，客戶程式可以送要求至多個伺服器來改善執行效率，甚至客戶程式可選用最迅速回應的資訊作為答案。

複雜的主從架構互動 (Cont.)

- 不限制一個伺服程式去執行再一層的主從式互動，即提供服務的伺服程式可以扮演另一個伺服程式的客戶：例如：當檔案伺服程式需要記錄檔案最後的存取時間，則檔案伺服程式可以變成時間伺服程式的客戶程式，待時間伺服程式回應後，再繼續處理原來檔案存取的要求。

互動和循環等待的關係

- 伺服器必須小心避免循環等待的發生。以上例而言，若時間伺服器也同使使用檔案伺服程式時，循環等待的狀態便可能會發生。
  - 例如：假設某一程式設計者被要求去修改時間伺服程式，所以它保留每個要求訊息的紀錄。若程式設計者讓時間伺服程式變成檔案伺服程式的客戶程式，即產生循環等待的結果。
- 要找出一隊互相等待的伺服程式並不難，但要找出一群分散各處相互等待的伺服程式就不那麼容易了，若每個伺服程式由不同程式設計者設計，那又更難了。

應用程式介面 (API)

- **API (Application Program Interface)** 定義介於應用程式與通訊協定軟體互動時執行的操作程式介面。
- 大部分定義 API 的系統程式是一組程序 (procedure) 的呼叫、宣告及參數設定，通常每個基本操作各有其對應的 API 程序，例如用來建立通訊連線的程序和用來傳送資料的程序。
- 通常通訊協定的標準並不規範與通訊協定互動所需的 API，而是規範一般必備的操作程序，並且允許各作業系統自行定義其 API 來執行這些操作程序。而 API 用來定義確實的函數名稱、參數和型別等細節。

Socket API

- Socket API 原本是 BSD UNIX 作業系統的一部份。當初是由美國政府出資，由 Berkley Univ. 所負責研發新版 UNIX 中，用在 TCP/IP 網際網路通訊協定。它是提供應用程式使用網路通訊的一組程序 (procedure)。
- 現在許多 OS 共用此介面，包括 Microsoft 的 Windows (Winsock) 和各種 UNIX 版本的 OS (Ex: Solaris)。
  - 通常 socket API 簡寫為 sockets

Sockets 和 Socket 程式庫

- 由於 Socket 是 BSD Unix OS 的一部份，為了在不同的作業系統上使用，許多廠商將它設計成一套 socket 程式庫，使其可以在非 Unix OS 或在一個沒有內建 socket 的 OS 上使用 socket API。
  - **內建式 socket 的程式碼**是 OS 的一部份；而 **socket 程式庫的程式碼**則是與應用程式連結在一起並且與應用程式使用相同的程式位址空間。
  - Socket 程式庫很少是完美無缺的，而且有時標準的 socket API 和 socket 程式庫間會有些微的差異 (如：錯誤例外處理)

Sockets 和 Socket 程式庫 (Cont.)

- 應用程式所呼叫的 socket 程序可以是 OS 的一個程序或是程式庫的一個程式。
- 當應用程式呼叫一程式庫的程序時，程式的控制權轉到程式庫的程式裡，而此程序通常會呼叫一次或多次 OS 內部程序來完成其作業。
  - 即 socket 程式庫的程式把底層作業系統函數包在裡面，而只提供使用者一個通用的 socket 介面。

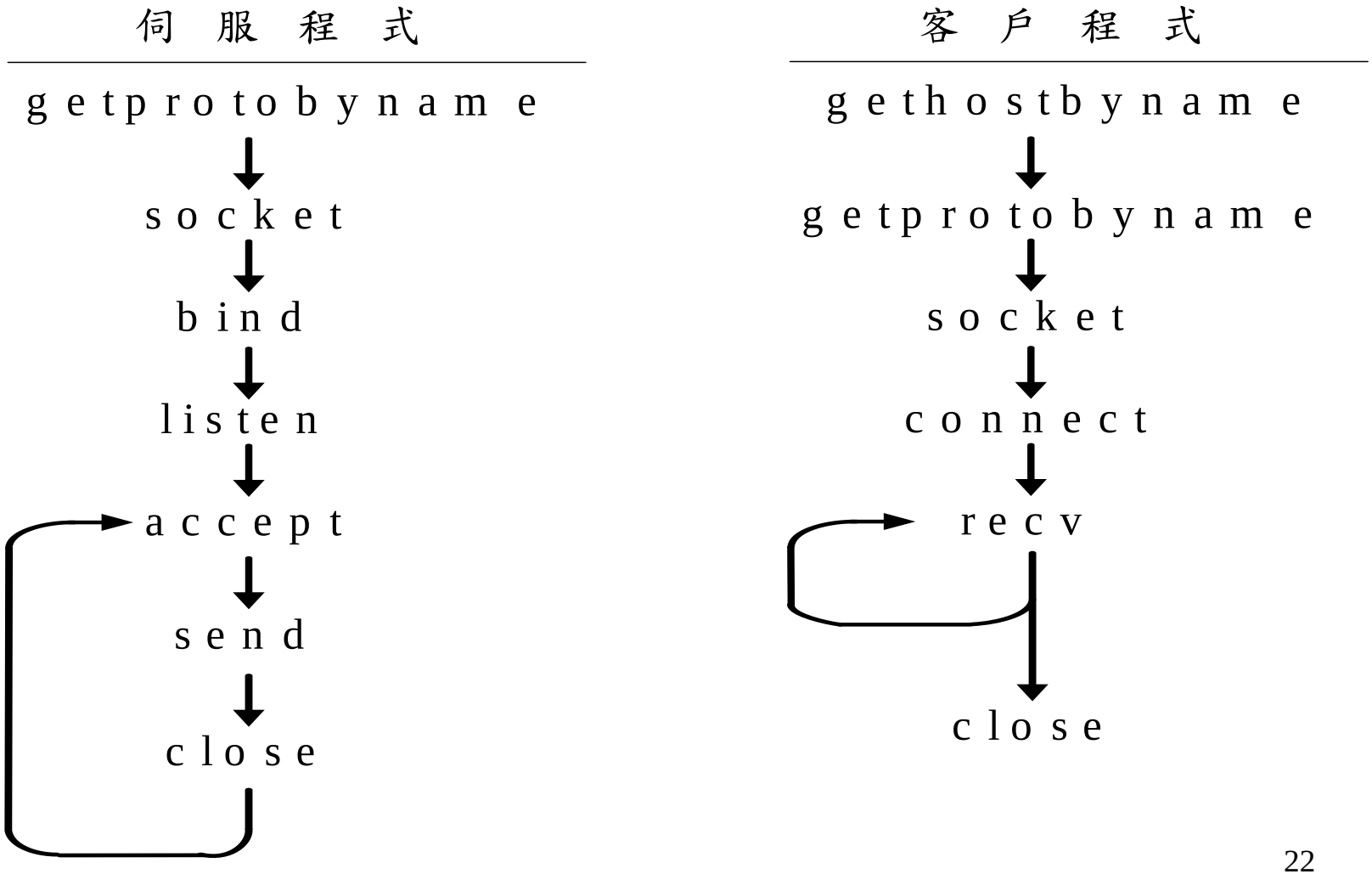
Socket 的通訊與 UNIX 的 I/O

- UNIX 將 socket 與其 I/O 的概念相結合，讓應用程式間的通訊就像在讀寫檔案一般容易。
- UNIX 的 I/O 採用 open-read-write-close 的作業模式。
  - 當應用程式開啟一檔案或週邊時，所呼叫的 open 系統程序會回傳一個 descriptor，其為一個整數值，用來當作此檔案的代號。在後續 read 或 write 時皆須利用此指標。

Socket 的通訊與 UNIX 的 I/O (Cont.)

- 在應用程式使用通訊協定做通訊時，需先要求作業系統產生一個 socket，系統將回傳一 descriptor 來代表此 socket。
  - 產生 socket 時，應用程式必須選擇所要使用的通訊協定種類，必須提供遠端主機的通訊協定位址，和指明此應用程式是 client 或 server 程式。因此，socket 必須有很多的參數和選擇設定 (option)，提供給應用程式選用。
- 基本上，應用程式先建立一 socket 後，再用其他的函數來指定其使用的相關細節。此種方法是大部分函數的參數最多不會超過三個，缺點是必須牢記較多的函數。

呼叫 socket 程序的順序



Socket API Functions

- 客戶或伺服程式可以呼叫 gethostname 取得本機的位址資訊
  - int gethostname(char *name, int namelen)
- struct protoent *getprotobyname(const char *name) returns a pointer to a protoent structure (protocol entity)，the members of which contain the fields of an entry in the network protocol database.
- 伺服程式呼叫 accept 收到一新連線後，可以呼叫 getpeername 來取得完整的遠端連線要求的 client 端程式之位址資訊。

Socket API Functions (Cont.)

- Socket：用來建立一個 socket，回傳一整數的 descriptor
  - descriptor = socket (protofamily，type，protocol)
  - Protofamily 參數用來指定此 socket 所用的通訊協定家族為何，例如，PF_INET (AF_INET) 代表 TCP/IP protocol。PF_DECnet 代表使用 DEC 公司所研發的通訊協定。
  - type 用來指定此 socket 所使用的通訊型別。兩種最常用的通訊型別是 connection-oriented data stream (文字代碼為 SOCK_STREAM) 和 connectionless message (文字代碼為 SOCK_DGRAM)。

Socket API Functions (Cont.)

– Protocol 參數用來指定此 socket 所使用的傳輸通訊協定。
  - Specifying a protocol of 0 causes socket() to use an unspecified default protocol appropriate for the requested socket type (e.g., TCP/IP).

Socket API Functions (Cont.)

- close：告知系統結束使用 socket (Winsock 使用 closesocket)。
  - close (socket)
  - socket 參數為欲關閉 socket 的 descriptor。
  - 若此 socket 是 connection-oriented 的傳輸通訊協定，close 在關掉 socket 之前會先結束網路的連線。
  - socket 關閉後立即停止使用，並將系統資源釋放掉，避免應用程式送出或收到更多的資料。

Socket API Functions (Cont.)

- bind：當 socket 建置時，並未宣告本機地址和遠端主機的位址。伺服程式需用 bind 程序來將 socket與本機上的一個 port 相關聯，之後 server 就可以在該port 監聽服務請求。
  - bind (socket，localaddr，addrlen)
  - socket 參數值為已建置但未指定 port number 的 descriptor
  - localaddr 為在名本機位址的資料結構之起始位置
  - addrlen 為本機位址的資料結構之長度

Socket API Functions (Cont.)

- 由於 socket 可適用於各種通訊協定，位址的格式因通訊協定而異。所以 socket API 定義了一個通用的資料結構來描述位址的格式，此通用格式稱為 **sockaddr** 資料結構。
  - struct sockaddr { /* IPv4 */

    u_char sa_len; /* total length of the address */

    u_char sa_family; /* family of the address */

    char sa_data[14]; /* the address itself */

    };

Socket API Functions (Cont.)

– struct socketaddr_in {

```
u_char sa_len;
u_char sa_family;
u_short sin_port;
struct inaddr sin_addr; /* IP address of the
                                    computer */
char sin_zero[8]; /*unused, for padding with the same
                          size as sockaddr */
};
```

Socket API Functions (Cont.)

- 利用計算机系统的多宿主能力（multi－homed capability），在只有一塊以太網卡的計算机上就能實现具有眾多IP地址的主机，而且每個 IP 地址還具有它們自己的 MAC 地址。這项技術可用于建立填充一大段地址空間的欺骗，且花费極低。
- 拿個例子來說明 Multi-homed IP Support 的用途. 假設有一部機器有兩個 IP/Hostname, 分別叫 ftp.heaven.net、ftp.hell.net。Multi-homed IP Support 就可以讓 Login 到不同 IP 的人感覺不出是同一部機器。也就是說，同樣是 Anonymous Login 不同 Hostname 進去之後的目錄可以不同。另外是可以提高網路連線的可靠度，如某個網路不通時，可經由第二條連線和網際網路連線。

Socket API Functions (Cont.)

- 當主機為多重位址主機 (multi-homed) 時，伺服程式只能從其選定的 IP 位址來收到服務要求。為解決此問題，socket API 可用 INADDR_ANY 來設定 IP 位址，代表伺服程式可以使用此電腦所有 IP 位址之同一個 port number 接收服務要求。

Socket API Functions (Cont.)

- listen：當伺服程式宣告其通訊協定 port number 後 (ex: bind)，可要求 OS 將此 socket 設為被動的等待 client 來聯繫的模式。
  - listen (socket，queuesize)
  - socket 參數值為已建置且已指定 port number 的 descriptor
  - queuesize 指定此 socket 的服務請求佇列 (request queue) 的最大值
  - 請求佇列一開始是空的，當客戶的請求抵達時，OS 會將其置入 request queue，直到伺服程式要擷取新的請求時，系統再從 request queue 中取出下一個服務請求給伺服程式。如果 request queue 滿了，則系統將捨棄新進的請求。

Socket API Functions (Cont.)

- accept：若伺服程式選用 connectionless 通訊傳輸協定，經過 socket 和 bind 之後，即進入等待訊息的狀態。但若伺服程式選用 connection-oriented 通訊傳輸協定，則在進入等待訊息狀態前需要額外的步驟：伺服程式需先呼叫 listen 程序使其 socket 進入等待連線模式，然後必須接受 client 程式連線的請求後，才能進入等待訊息狀態。一旦連線被接受後，伺服程式即可和客戶程式相互通訊，直到通訊完畢再關閉連線。

Socket API Functions (Cont.)

- 使用 connection-oriented 傳輸通訊協定的伺服程式必須呼叫 accept 來接受連線要求。若有 client 的連線請求在 request queue 中，則 accept 程序立即回應此請求。若尚無 client 連線請求，則系統會將伺服程式的呼叫暫停執行，直到有新的 client 連線請求抵達 server 端。
  - newsock=accept (socket，caddress，caddresslen)
  - socket 為已完成建置且已指定本機位址的 descriptor
  - caddress 為 sockaddr 資料結構的起始位置
  - caddresslen 為位址的長度

Socket API Functions (Cont.)

- accept 會將要求連線的客戶端程式之位址填入 caddress 的資料結構中，並將其位址長度填至 caddresslen 所指的整數變數內。最後再建立一個與客戶程式通訊專用的新 socket，然後再將新 socket descriptor 回傳給伺服程式。
- 伺服程式即可利用此新的 socket descriptor 和客戶程式相互通訊，當通訊完畢時再關閉此 socket。也就是說伺服程式原先使用的 socket 依然還在，伺服程式在和客戶程式通訊完關閉新的 socket 後，伺服程式仍可以用原來的 socket 繼續接受下一個客戶程式的連線請求。

Socket API Functions (Cont.)

- connect：客戶程式利用 connect 與指定的伺服程式建立連線。
  - connect (socket，saddress，saddresslen)
  - socket 為用來連線的客戶端之 socket descriptor
  - saddress 為一指向記錄伺服端位址和通訊協定埠號碼之 sockaddr 資料結構的位址
  - saddresslen 為 sockaddr 資料結構的長度
- 當使用 connection-oriented 傳輸協定如 TCP 時，connect 程序啟動和其所指定的伺服程式建立傳輸層的連線。基本上，客戶程式呼叫 connect 程序和已呼叫 accept 程序的伺服程式做連線。

Socket API Functions (Cont.)

- 當客戶使用 connectionless 傳輸協定時，也一樣可以呼叫 connect 程序，但實際上並沒有啟動連線作業或送任何封包到網路上。反而只是將 socket 直接註記成已連線的狀態並且登記伺服程式的位址在 socket 中。
- 為何要在 connectionless 傳輸通訊協定中使用 connect 程序呢？因為 connectionless 協定要求發送者在每個訊息上均填入目的地位址，但大部分的應用程式都是一個 client 程式和某一個伺服程式通訊，因此所有的訊息都有相同的目的地位址。為簡便起見，客戶只需宣告一次伺服程式位址，不用每個訊息都要再重複指定其位址，方便連續傳送訊息。

Socket API Functions (Cont.)

- send：客戶和伺服程式都需要發送資訊。若 socket 已連線成功，可用 send 程序來發送資料。
  - send (socket，*data，length，flags)
  - data 為指向要發送資料的起始位址
  - length 為所發送資料的總長度
  - flags 為發送資料的型別，flags參數值可能為 0 或是下面三個值之一，一般設定為 0。
    - MSG-OOB (out of band)：傳送或接數緊急資料
    - MSG_PEEK：MSG_PEEK旗幟讓呼叫者查看資料是否可以讀取，不需在recv或recvfrom 返回後 才讓系統將資料刪除。
    - MSG_DONTROUTE：目的地是在 local，不需查 routing table

Socket API Functions (Cont.)

- sendto 和 sendmsg 都是提供給 client 和 server 程式使用 connectionless 傳輸協定傳送訊息使用的。
  - sendto (socket，*data，length，flags，*destaddress，addresslen)
  - sendmsg 和 sendto 有相同的功能，但是它將參數簡化成一個資料結構。
  - sendmsg (socket，*msgstruct，flags)

Socket API Functions (Cont.)

- msgstruct 為指向 msghdr 的資料結構
- struct msghdr {

```
  void *msg_name; /* protocol address */
  socklen_t msg_namelen; /* size of protocol address */
  struct iovec *msgiov; /* scatter/gather array */
  size_t msg_iovlen; /* # elements in msg_iov */
  void *msg_control; /* ancillary data */
  socketlen_t msg_controllen; /* length of ancillary data
   */
  int msg_flags; /* flags returned by recvmsg() */
};
```

Socket API Functions (Cont.)

- msg_name points to a data structure which the sender stores the destination address
- msgiov 和 msg_iovlen 描述 array of input/output buffer (buffer 是 iovec structure)
  - struct iovec {
    void *iov_base; /* starting address of buffer */
    size_t iov_len;
    }
- msg_control 和 msg_controllen 描述額外資料的位置和大小 (E.g., hop limit, next hop address)。
- msg_flags 只有在 recvmsg 有用，在 sendmsg 無用。

Socket API Functions (Cont.)

- recv：客戶和伺服程式都需要接收對方發送的資料。應用程式能呼叫 recv 從**已連線**的 socket 中接收資料。
  - recv (socket，*buffer，length，flags)
    - buffer 指向存放接收資料的起始位址
    - length 為 buffer 總長度
    - flags：包含 MSG_OOB，MSG_PEEK…
- recvfrom：若 socket 尚未連線，且被用來接收任何一個 client 程式的訊息時，應用程式可以利用 recvfrom 程序來接收訊息並儲存發送者的位址。

Socket API Functions (Cont.)

- recvfrom (socket，*buffer，length，flags，*sndaddress，*saddrlen)
  - 最後兩個參數用來記錄訊息的發送者位址，其中 sndaddress 為指向系統記錄發送者位址之資料結構位址。saddrlen 為位址資料結構的長度。

- recvfrom 程序的發送者位址記錄正是 sendto 程序的目的地位址。因此若應用程式使用 recvfrom 來接收新訊息，其回覆訊息便很簡單，只要將其記錄的位址當作目的地位址回傳訊息即可。

Socket API Functions (Cont.)

- sendmsg：對應 recvmsg 程序，運作和 recvfrom 相同，但參數較少。
  - recvmsg (socket，*msgstruct，flags)
  - msgstruct 指向用來存放新進訊息的儲存位置及發送端的位址之資料結構。
  - 和 sendmsg 剛好是一對。

Socket 的讀與寫

- Socket 也允許應用程式使用 read/write 來傳送/接收資料。read 與 write 使用三個參數：socket descriptor、指標指向存放資料的記憶體起始位置和暫存記憶體空間的大小
- read 與 write 必須與已連線的 socket 合用。
- 使用 read/write 程序最大的好處是具有通用性，可以撰寫一個從 descriptor 讀出和寫入資料的應用程式，而不用去考量此 descriptor 所代表的是一個檔案或是一個 socket。因此程式設計員可以利用本機的檔案做為客戶與伺服程式的測試，而其缺點則是會增加原本 I/O 的複雜度。

其他的 socket 程序

- socket 程序有許多的參數及 options，setsockopt 設定 socket option 值，setsockopt 為讀取 socket option 值。
  - setsockopt(socket，level，optname，optval，optlen)
  - level= SOL_SOCK or TCP level
  - options：SO_BROADCAST (broadcast)，SO_KEEPALIVE (keeps connections active by enabling the periodic transmission of messages)

其他的 socket 程序 (Cont.)

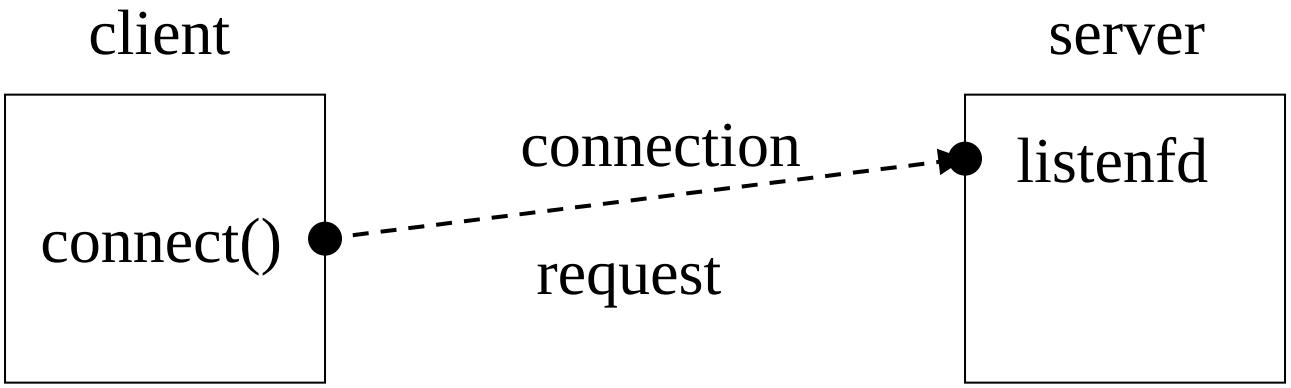
- gethostbyaddr 與 gethostbyname 相反，將 IP 位址轉成對映的電腦名稱。

Socket、thread 和繼承性

- 新的 thread 被建立時，應從產生它的 thread 身上繼承一份所有已開啟的 socket。
- Socket 使用 reference count機制，當一個 socket 被建立時，系統就把 socket 的 reference count 設為 1，當這個 reference count大於零的時候，socket都會存在。
  - 當程式產生新的 process/thread 時，會把 parent 和 child process 的 reference count 都加 1。

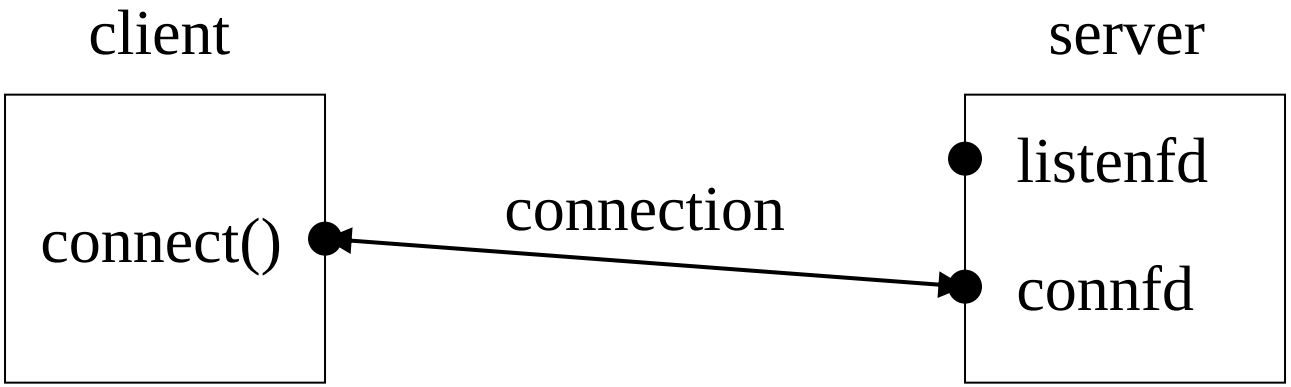
Socket、thread 和繼承性 (Cont.)

- Status of client-server before call to accept returns



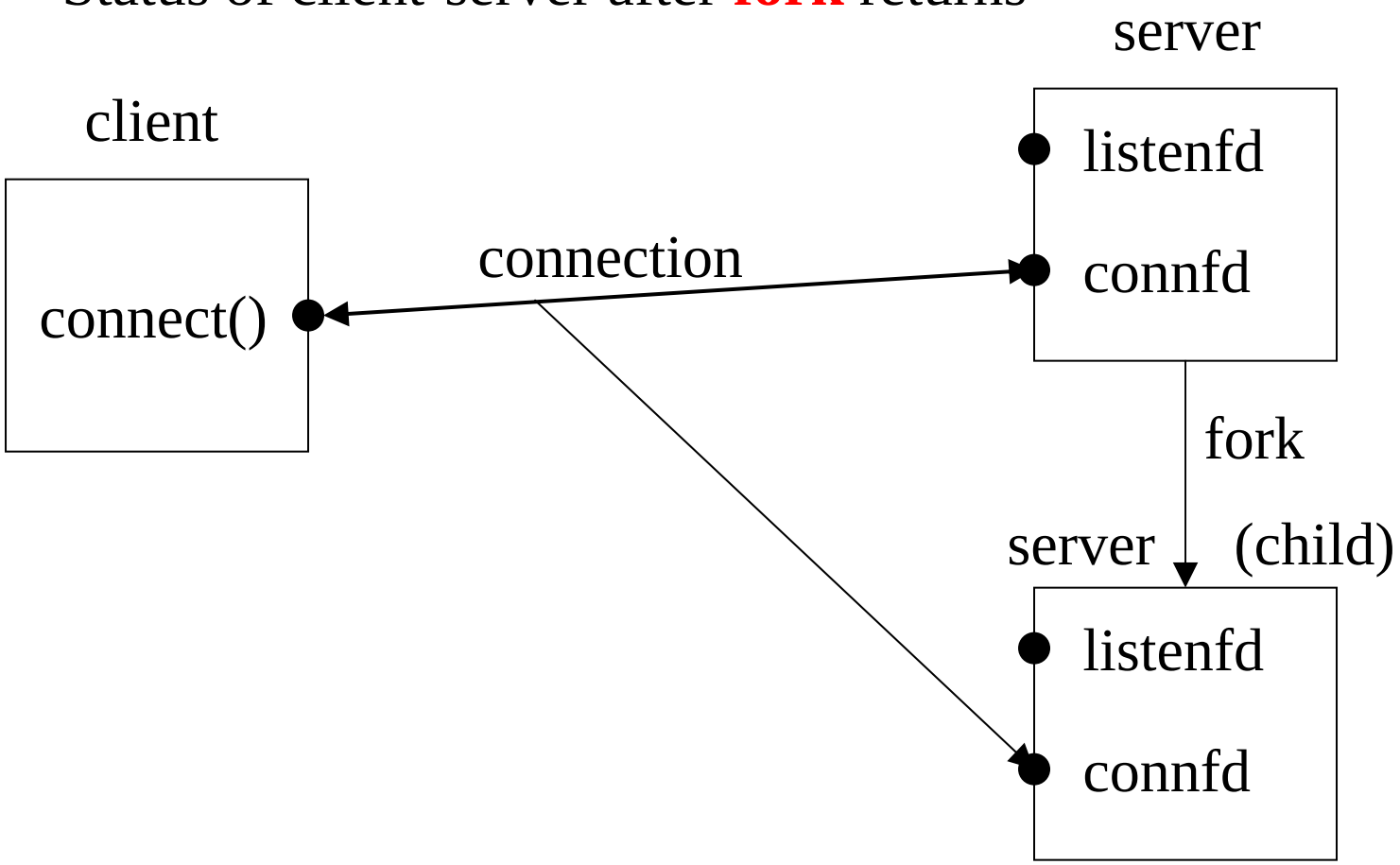
Socket、thread 和繼承性 (Cont.)

- Status of client-server after return from accept



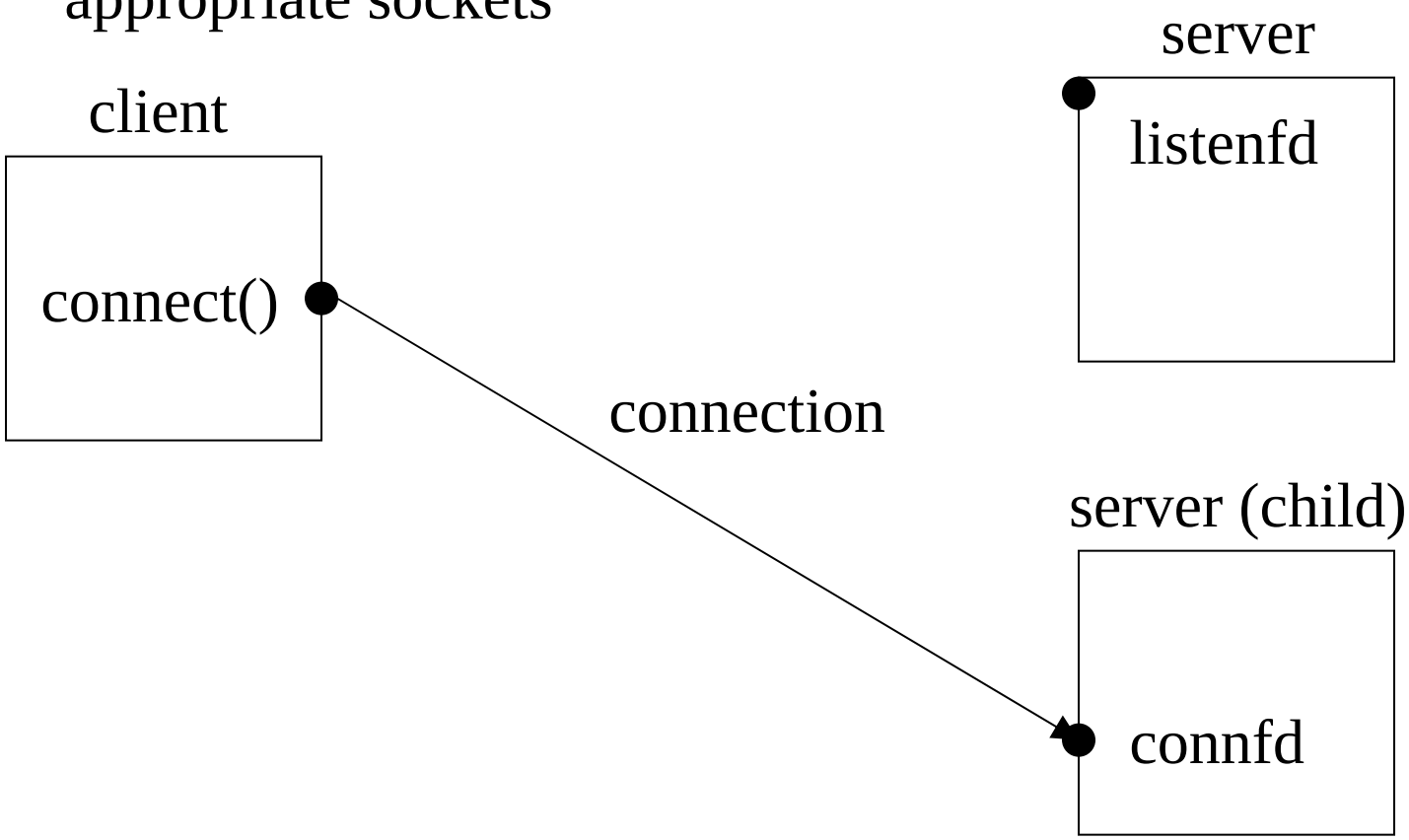
Socket、thread 和繼承性 (Cont.)

- Status of client-server after **fork** returns



Socket、thread 和繼承性 (Cont.)

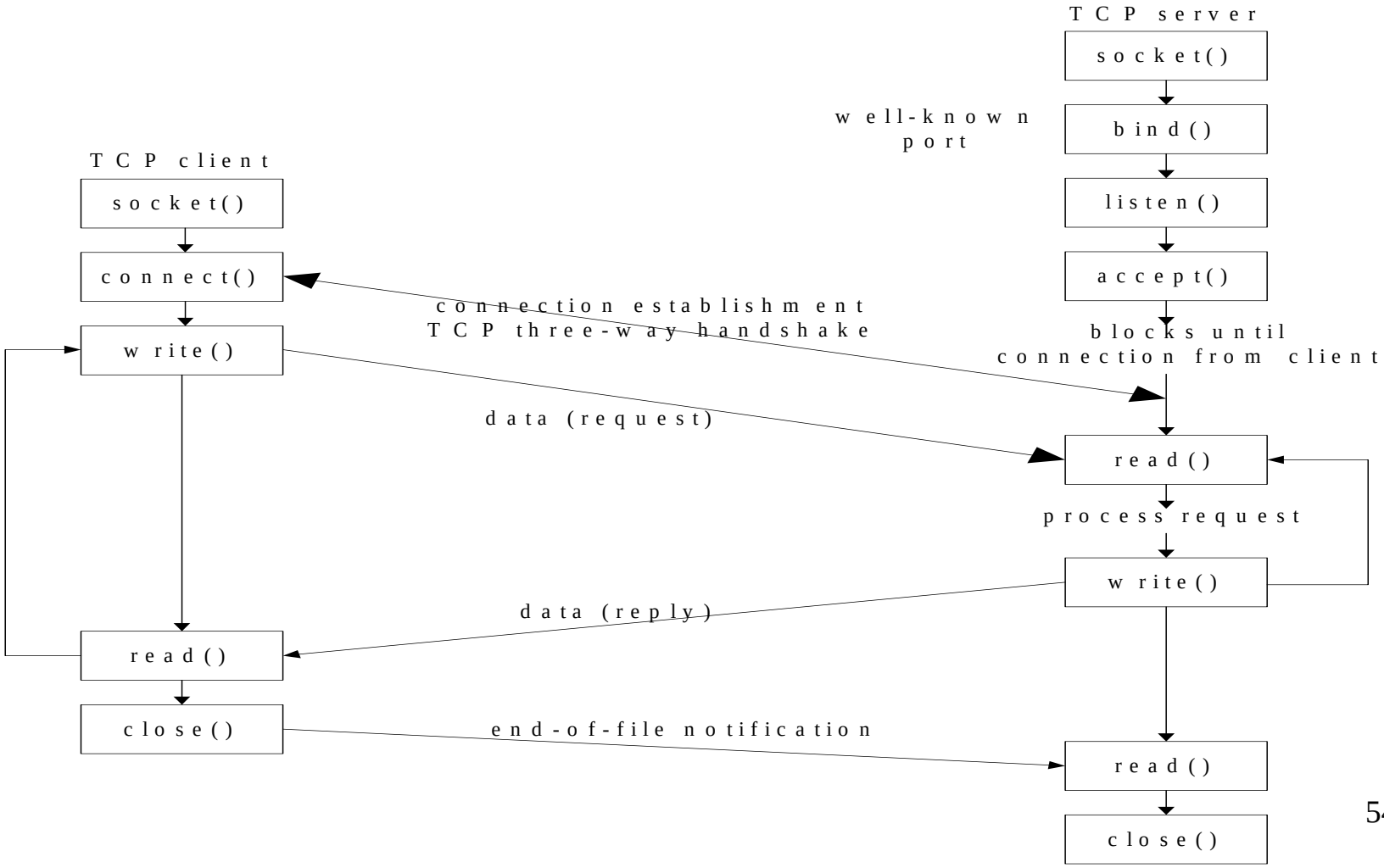
- Status of client-server after parent and child close appropriate sockets



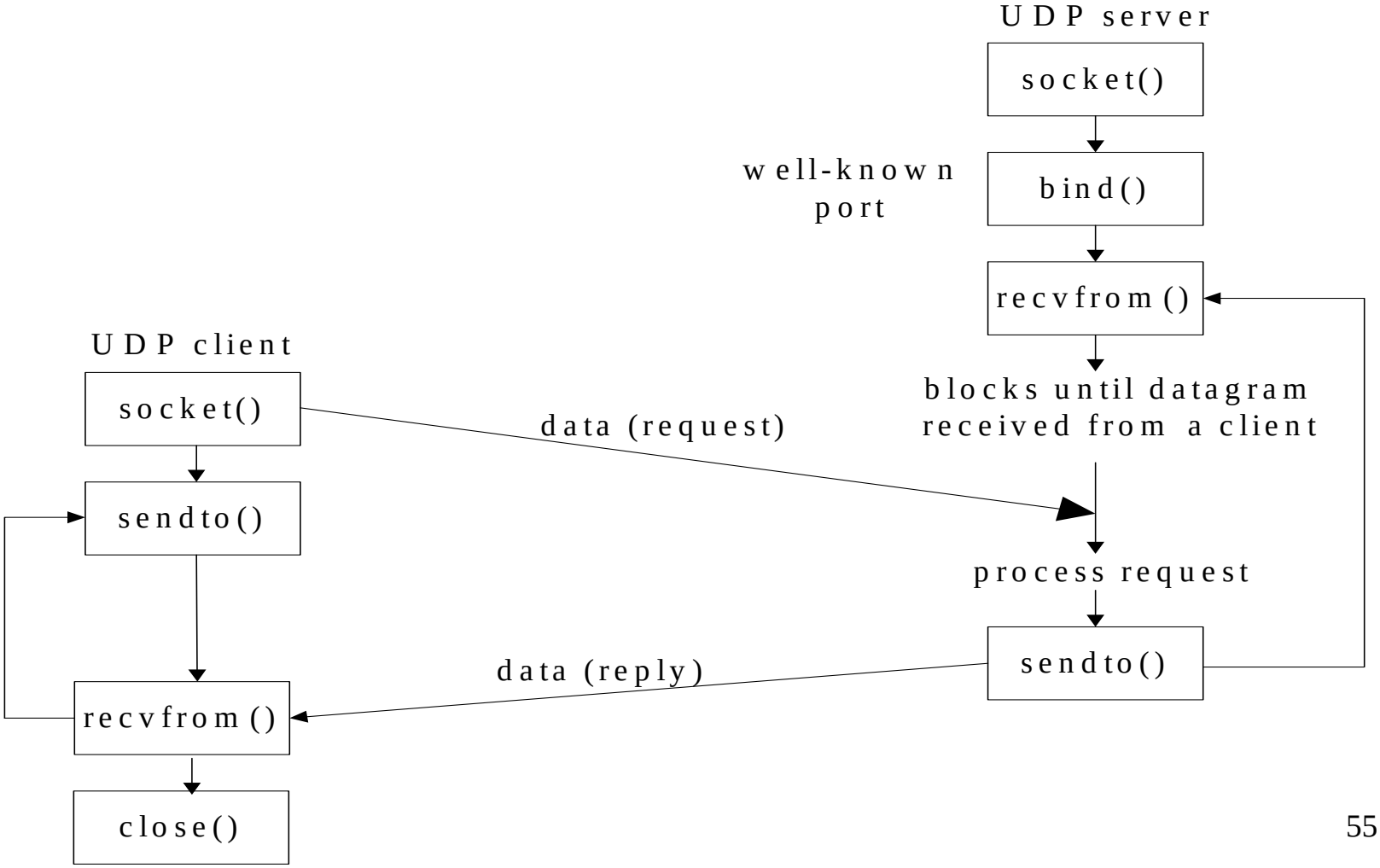
Socket、thread 和繼承性 (Cont.)

- 當某一 process/thread 呼叫 socket 的 close 程序時，系統將其 reference count 值減一，並且從 thread 清單去除。
- 若 thread 在關閉 socket 前結束，系統央會把 thread 開啟的所有 socket，執行 close 程序。

Socket Functions for Elementary TCP Client-Server



Socket Functions for UDP Client-Server



An Example：A Simple Daytime Client

- #include “unp.h”

```
int
main (int argc, char **argv)
{
   int sockfd, n;
   char recvline[MAXLINE+1];
   struct sockaddr_in servaddr;

   if(argc!=2)
      err_quit(“usage: a.out <IPaddress>”);
```

An Example：A Simple Daytime Client (Cont.)

```
if((sockfd=socket(AF_INET, SOCK_STREAM,0)) < 0)
   err_sys(“socket error”);

bzero(&servaddr,sizeof(servaddr));
servaddr.sin_family=AF_INET;
servaddr.sin_port=htons(13);
if(inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0)
   err_quit(“inet_ption error for %s”,argv[1]);

if(connect(sockfd, (SA *) &servaddr, sizeof(servaddr)) < 0)
   err_sys(“connect error”);
```

An Example：A Simple Daytime Client (Cont.)

```
  while ((n=read(sockfd, recvline, MAXLINE)) >0) {
    recvline[n]=0; /* null terminate */
    if (fputs(recvline, stdout) = = EOF)
      err_sys("fputs error");
  }
  if (n>0)
    error_sys("read error");
  exit(0);
}
```

An Example：A Simple Daytime Server

- #include “unp.h”
  #include <time.h>

  int
  main(int argc, char **argv)
  {
    int listenid, connfd;
    struct sockaddr_in servaddr;
    char buf[MAXLINE];
    time_t ticks;

    listenid=socket(AF_INET, SOCK_STREAM,0);

An Example：A Simple Daytime Server (Cont.)

```
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family=AF_INET;
servaddr.sin_addr.s_addr=htonl(INADDR_ANY);
servaddr.sin_port=htons(13);

bind(listenid, (SA *)&servaddr, sizeof(servaddr));

listen(listenfd, LISTENQ);

for( ; ; ) {
   connfd=accept(listenfd, (SA *)NULL, NULL);
```

An Example：A Simple Daytime Server (Cont.)

```
        ticks=time(NULL);
        snprintf(buff, sizeof(buff), “%.24s\r\n”, ctime(&ticks));
        write(connfd, buff, strlen(buff));

        close(connfd);
    }
}
```