第九章 RPC 高階程式介面

9-1 RPC 簡介

我們在上一章裡介紹低階（Low-level）網路程式介面 - Socket，雖然利用 Socket Library 來開發程式，使用者不用理會有關網路的運作方式，但還是必須考慮到有關傳輸埠口的連接方式，和較低層次的傳輸動作（read() 與 write()）。如果我們能夠在 Socket 的基本架構上，再開發出更高階的程式介面，讓使用者呼叫遠端電腦上的程式，就如同呼叫本地程式一樣容易，對使用者而言，宛如沒有網路存在的方便性，這就是發展**『遠端程式呼叫』（Remote Procedure Call, RPC）**的基本理念。

早期 RPC 是 Sun 公司在**『網路檔案系統』（Network File System, NFS）**上發展出的技術，其功能在於提供不同系統之間的檔案存取工具。近年來許多軟體廠商沿用 RPC 技術，應用在不同領域上的網路存取，最常見的是**『資料庫伺服器』（Database Server）**所提供的網路之間資料處理工具，譬如，資料的查詢（Query）、插入（Insert）、更新（Update）、刪除（Delete）等等，這些處理工具都是透過 RPC 來實現。

一般在發展應用系統時，我們會將一些專屬功能程式，另外編寫成副程式，來讓其它應用程式呼叫，也稱為**『程序呼叫』（Procedure Call）**，其運作方式如圖 9-1 所示。一個專屬程序也許會給予一個特殊功能，譬如一般資料庫系統，我們會將有關資料的查詢、更新、插入等等的動作製作成專屬程序，其它應用程式只要呼叫該程序並給予適當參數，就可以得到所需要的資訊。如果專屬程序和應用程式都存在同一部主機上，問題就比較簡單，一般也稱之為**『本地程序呼叫』（Local Procedure Call, LRC）**。但近來網路應用非常普遍，常需要將某些專屬功能以一個獨立的設備來實現，以增加系統的效率。譬如，時尚非常流行將資料庫系統以一專屬設備來服務，也就是常見到的**『資料庫伺服器』（Database Server）**，它所提供的專屬程式（查詢、更新、插入等）就必須能夠讓遠端電腦（或客戶端）呼叫，其運作方式如圖 9-2 所示。一部專屬主機能提供各種服務程式，讓遠端電腦透過網路來查詢，此工作模式稱之為**『遠端程序呼叫』（Remote Procedure Call, RPC）**，這就是網路上的高階程式介面。

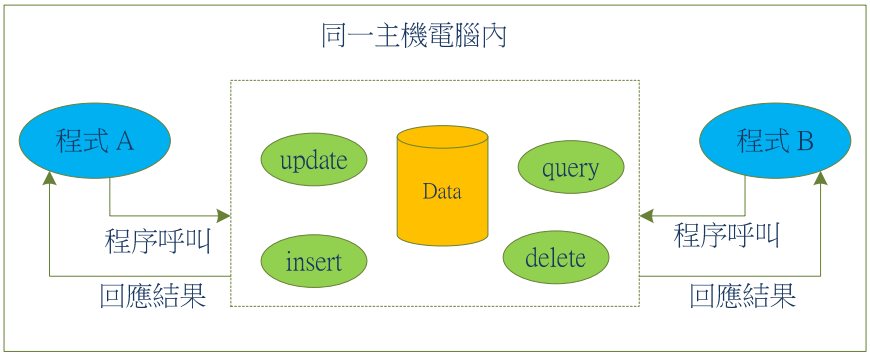

圖 9-1 本地程序呼叫的運作方式

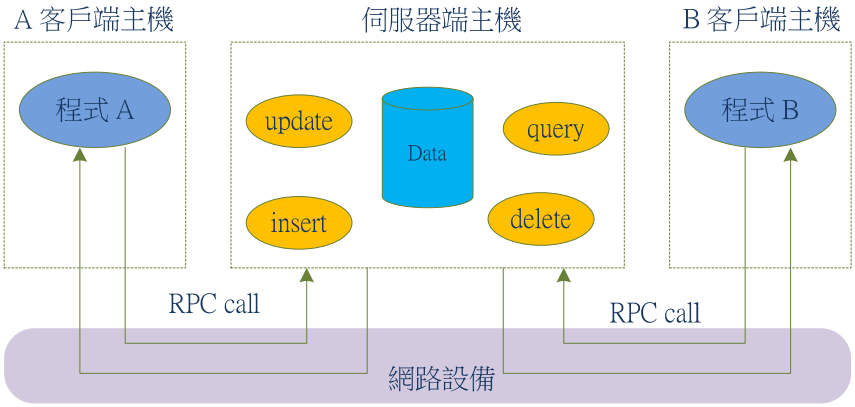

圖 9-2 遠端程序呼叫的運作方式

9-2 RPC 協定架構

我們用 OSI 參考模型來介紹 RPC 的協定架構，也許會讓讀者比較容易瞭解，說明如下：(如圖 9-3 所示)

※ **應用層**：也就是我們一般發展的應用程式，譬如，Client/Server 資料庫伺服器。

※ **表現層**：在不同主機上，對於抽象資料型態 (Abstract Data Type) 的表示法也許會有所不同。如果 Client/Server 的資料表示法不同，當它們執行呼叫遠端程序時，很容易發生嚴重的錯誤，因此，程序呼叫之前必須將所傳遞的參數以標準格式表示，才不會發生錯誤。目前一般應用環境大多以『外部資料表示法』(eXternal Data Representation, XDR) 來表示資料型態，但近年來也有許多環境開始以 ASN.1 表示 (請參考 8-10 節介紹)。

※ **交談層**：遠端程序呼叫實現在這層次裡。

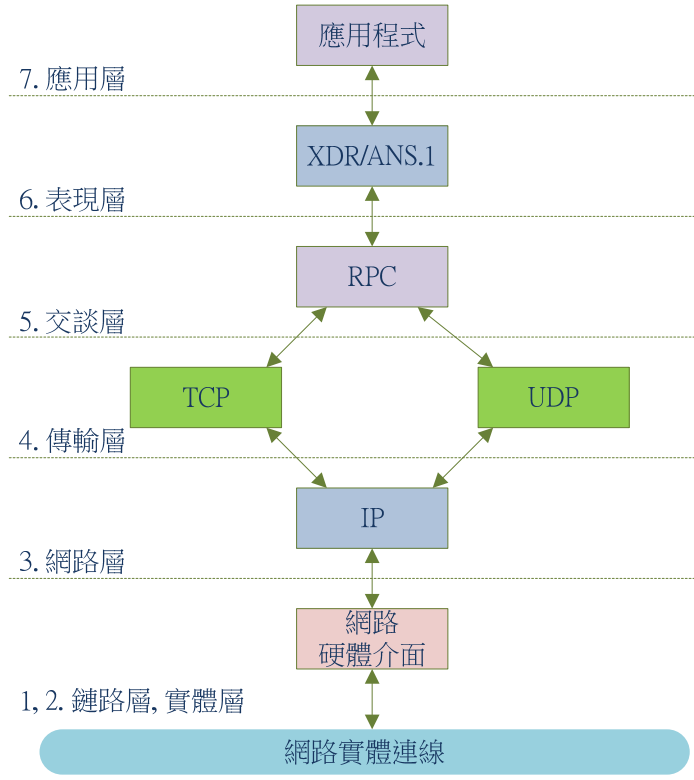

圖 9-3 RPC 系統與 OSI 參考模型

※ **傳輸層：**可採用 TCP 或 IP 協定，也就是使用連接導向（虛擬電路）或非連接方式（電報傳輸）。

※ **網路層：**採用 IP 協定。

※ **鏈路層與實體層：**架設在一般區域網路或 Internet 網路上。

RPC 程式介面工具最主要是架設在傳輸層上，但不同傳輸層介面也會引起系統設計的不同。一般傳輸層介面大略可區分為：傳輸獨立（Transport-Independent, TI）介面和 Socket 介面兩種，前者稱之為 TIRPC；而後者稱之為 Socket-based RPC。TIRPC 是架設在 Network Selection 系統上，它是將有關網路上的伺服器、服務程式、網路位址、傳輸埠口等等都宣告成符號層次（Symbolic Level），而以 rpcbind() 程序來管理呼叫。Socket-based RPC 較為單純，直接連接到 Socket 傳輸埠口上，以 portmap() 程序取代 rpcbind()。雖然 Socket-based RPC 在製作上比較容易，但沒有 TIRPC 的變通性高，因為 TIRPC 將網路環境都以資料型態表示，不論現有的本地程序呼叫轉換成遠端程式呼叫，或網路環境變更，TIRPC 的處理都比 Socket-based RPC 較為容易。又從另一角度來看，TIRPC 比較適合一般範圍較小的區域網路使用；然而 Internet 網路上較適合簡單連結方式的 Socket-based RPC。本章為了延續上一章的介紹，而採用 Socket-based RPC 來介紹，至於 TIRPC 因限於篇幅不再另述。

無論 RPC 採用 Socket-based 或是 TIRPC，在連線方面也可區分為連接導向（TCP）和非連接方式（UDP），它們的運作方式如圖 10-4 與 10-5 所示。至於 RPC 連線應該選用 TCP 或 UDP 協定？這就得好好考慮。雖然 TCP 連線可保證訊息是否安全到達目的，並且可將連續大量的資料在預期時間內連續送達目的，但也有其缺點，譬如，每一連線會佔用系統資源，一般系統都會限制連線的數量，才不會使整個系統癱瘓掉。另一方面也要考慮到，如果傳輸資料過於短少，則先建立連線再傳送資料的過程就不需要了。UDP 連線雖然不能保證訊息是否安全到達，但也有其時效性，對於需要快速傳輸或較少量的資料而言，的確是個很好的傳輸工具，更何況它所佔用的資源最少，又不會受到系統連線數目的限制。但 UDP 最大的缺點是傳送端發送訊號時，可能會因為認為對方沒有收到訊息，而重複傳送同樣的訊息。當接收端收到重複訊息時，得有能力判斷而將之拋棄，才不至於發生錯誤。因此，在下面情況允許範圍內，可考慮使用 UDP：

- 遠端程序是否屬於『**等冪性**』（**Idempotent**），也就是遠端程序可連續接收同一訊息，而不會出問題。
- 呼叫訊息和回應訊息都可以在一個 UDP 封包內包裝完成。
- 伺服器需要多重客戶端要求，或客戶端需要多重伺服器要求的遠端呼叫程序。因為 UDP 不受系統連線數目的限制，也就是所使用的系統資源較少，因此可同時提供多重連線傳輸資料。

另外，如欲使用 TCP 連線的考慮因素如下：

(1) 對可靠性要求較高的遠端程序呼叫。

(2) 遠端程序不屬於『**等冪性**』的。

(3) 呼叫程序或回應訊息無法用一個 UDP 封包包裝。

由上述我們可以瞭解，UDP 協定在許多應用範圍也非常適合，譬如，DNS 伺服器之間的查詢。早期 Sun 發展 RPC 是應用在區域網路上的 NFS 系統上，一般區域網路範圍較小，封包在網路上不會跨越太多的路由端點，IP 封包在網路上不會有太多的延遲現象，因此，在 NFS 系統上是利用 UDP 來製作 RPC。但目前有許多應用發展在 Internet 網路上，通訊之間也許會經過較多的路由端點，對於傳輸的可靠性就較為薄弱，因此，大多建議使用 TCP 協定來銜接 RPC 程式。

9-3 RPC 運作方式

RPC 運作方式可區分為兩大部份：RPC 連線方式和 RPC 程序呼叫方式，以下分別介紹之。

9-3-1 RPC 連線方式

圖 9-4 是 RPC 連線方式的運作圖，它的基本架構是伺服端有一個超級服務程式（Portmap）監督 RPC 連線要求，並告知 Client 端遠端程序呼叫的程式是位於哪一個埠口，運作方式如下：

(1) 當伺服程式（RPC Server）啟動時，便向 Portmap 註冊本身的傳輸埠口位址（含 IP 位址）和伺服程式名稱。

(2) 客戶端如欲呼叫遠端程式，首先向伺服器的 Portmap 查詢遠端程式所連接的埠口位址。

(3) 客戶端得到遠端程式的埠口位址後，便可以直接透過該位址連結到遠端程式。

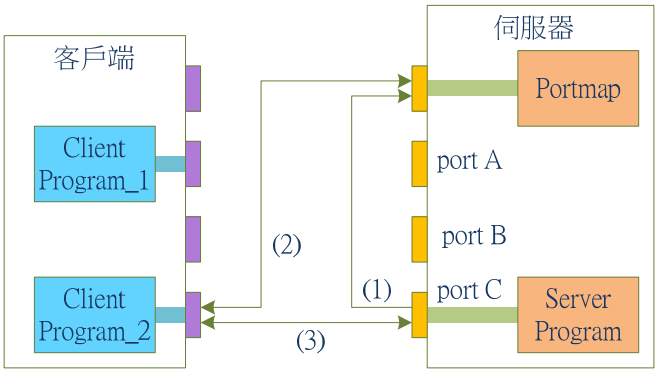

圖 9-4 RPC 連線方式

由以上的敘述得知，我們需要一個 Well-known 的傳輸埠口（Portmap），來隨時監督等待 Client 的連線要求，並告知服務程式的位址，這個 Well-known 傳輸埠口一般都設定在 111/udp 或 111/tcp 埠口上，它是屬於 /etc/services 描述檔中的 sunrpc 服務。至於 /etc/services 部分檔案內容如下：

```
# cat /etc/services
pop3        110/tcp          pop-3          # POP version 3
pop3        110/udp          pop-3
sunrpc      111/tcp          portmapper     # RPC 4.0 portmapper TCP
sunrpc      111/udp          portmapper     # RPC 4.0 portmapper UDP
auth        113/tcp          authentication tap ident
```

當 portmapper 收到 RPC 連線要求時，會依照 /etc/rpc 內所描述的**『程式號碼』（Program Number）**，尋找相對應的埠口位址。一般系統對於程式號碼的編排如下：

- **號碼 0x0 ~ 0x1FFFFFFF**：此範圍內保留給 Sun Microsystem 現有的呼叫程式使用。一般使用者不可使用此範圍號碼。
- **號碼 0x20000000 ~ 0x3FFFFFFF**：此範圍給一般使用者規劃自行的遠端程序呼叫。
- **號碼 0x40000000 ~ 0x5FFFFFFF**：暫時性（Transient）使用。
- **號碼 0x60000000 ~ 0xFFFFFFFF**：保留未用。

有關程式號碼的編排放置於 /etc/rpc 檔案內，其內容如下：(部分範例內容)

```
# cat /etc/rpc
#ident    "@(#)rpc1.11       95/07/14 SMI"      /* SVr4.0 1.2      */
#
#rpc
#
portmapper          100000    portmap sunrpc rpcbind
rstatd               100001    rstat rup perfmeter rstat_svc
rusersd              100002    rusers
nfs                  100003    nfsprog
ypserv               100004    ypprog
mountd               100005    mount showmount
ypbind               100007
walld                100008    rwall shutdown
yppasswdd            100009    yppasswd
etherstatd           100010    etherstat
selection_svc        100015    selnsvc
database_svc          100016

bwnfsd               545580417
fypxfrd              600100069 freebsd-ypxfrd
sched                 100019
```

9-3-2 RPC 程序呼叫方式

圖 9-5 為 RPC 程序呼叫方式，也就是在 RPC 連線建立後如何執行程序呼叫的運作。從客戶端呼叫遠端程序到收取伺服端回應訊息，其中經過了 Client Stub 和 Server Stub 兩個基礎檔（Stub），Stub 主要的功能是處理連線雙方資料格式的轉換，一般標準傳輸資料格式都採用 XDR 格式。

當 Client 執行某一程序呼叫（clnt_call()），它經由 Client Stub 時，也許會產生許多網路的系統呼叫來實現它，這些系統呼叫主要是讓 Client Stub 和 Server Stub 之間通訊使用，另外 Client Stub 也會將呼叫程序中的參數轉換成 XDR 格式。Server Stub 收到 Client 端訊息後，將呼叫程序的參數由 XDR 轉換成 Server 端主機的資料格式，再呼叫 RPC Server 上的遠端程序（如範例中的 add_1_svc()）。遠端程序執行完成後，再將結果傳送給 Server Stub，Server Stub 又將傳送回去的參數轉換成 XDR 格式，並從事下層協定的呼叫來連接與傳遞給 Client Stub，Client Stub 再將參數由 XDR 格式轉換成 Client 端主機的格式，並傳遞給 RPC Client。在整個呼叫傳遞過程之中，都屬於同步傳輸方式（也就是 Blocking 方式），RPC Client 端會一直等待到有回應，才會再執行下一個步驟，否則便以計時器溢時（Time out）方式來中斷它。

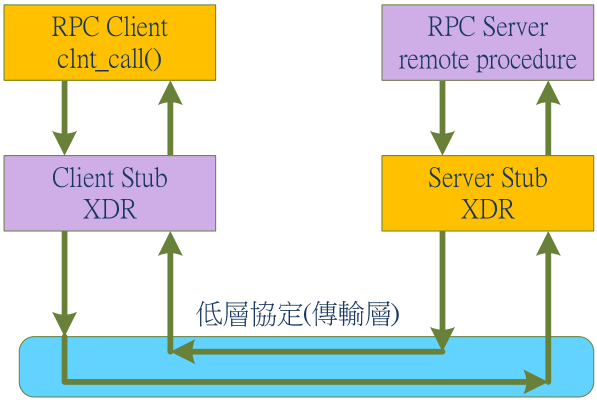

圖 9-5 RPC 程序呼叫方式

9-4 RPC 程式開發

欲開發一個 RPC 程式，首先必須將一些遠端程序組合一個程式，它的做法是設定一個程式，並給予一個**『程式號碼』（Program Number）**來作為識別，一個程式號碼包含若干個程式版本，每一程式版本也給予一個**『版本號碼』（Version Number）**以作為識別，同樣的，每一版本號碼也包含若干個遠端程序，每一遠端程序也給予一個**『程序號碼』（Procedure Number）**來作為識別。這表示程式號碼定義一組程序，版本號碼讓使用者隨時擴充程式功能，而程序號碼描述程式內的程序。

然而 RPC 程式是一個高階程式介面，當我們開發 RPC 程式時，有關低階介面（如 Socket）的處理程序必需儘量去簡化它。早期 Sun 為了簡化程式設計，開發一個 RPC 程式的產生器，稱之為**『rpcgen』**。rpcgen 的規範為 RFC 1057（新的工具為 new-rpcgen 功能比 rpcgen 強），它提供一系列的規格檔（Specification File）格式（9-7 節介紹）。規格檔經過 rpcgen 編譯後，會自動

產生『伺服基礎檔』(Server Stub) 和『客戶基礎檔』(Client Stub)，並依照選項 (Option) 來產生認證 (Authentication) 交換檔和資料表示檔 (XDR) (RFC 1014)。客戶端和伺服端程式只要連結這些檔案，就不需要考慮有關下層連接介面和資料格式的表示法。因此，我們可將 rpcgen 的功能歸類如下：

- rpcgen 是一個編譯程式 (Compiler)，它可依照規格檔來定義遠端程序呼叫的介面，並產生伺服端和客戶端的基礎檔。
- 可照選項來決定是否包含共通資料表示法 (XDR) 和認證系統 (Kerberos)。
- 並連結相關 RPC 庫存函數 (RPC Library)。

我們用圖 9-6 來說明發展 RPC 程式的過程，如以下步驟：

(1) 編寫 RPC 規格檔：規格檔中描述伺服程式和客戶程式之間的交換資料格式，以及呼叫程序的名稱，並指定遠端程式號碼與程式版本 (如 math.x)。規格檔的編寫格式如 9-7 節說明。

(2) 利用 rpcgen 編譯程式產生四個共用檔案：標頭檔 (math.h)、Server Stub (math_svc.c)、Client Stub (math_clnt.c) 與資料格式轉換檔 (math_xdr.c)。Server Stub 與 Client Stub 兩個檔案一般都不希望使用者去更改它，而只當作編譯時連結使用。標頭檔內定義有關程序呼叫的參數格式和程序名稱，一般也不希望使用者更改，作為遠端程序和客戶端程式原始檔的包含檔 (Include File) 使用。不同主機上的客戶端程式，如想要呼叫該遠端程序，都必須要依此前頭檔來編寫程式，因此必須將其複製到不同的客戶端電腦上。rpcgen 的命令格式如下：(以 math.x 為範例)

```
$ rpcgen math.x
```

(3) 編寫遠端程序：必須將共通前頭檔加入 (如，#include <math.h>)，並依照遠端程序功能編寫程式。編譯時必須將 Server Stub 加入，範例如下：(lnsl 中的 l 是 L 的小寫)

```
$ cc –o math_proc math_proc.c math_svc.c math_xdr.c  –lns1
```

(4) 編寫客戶端主程式：客戶端也是依照共通標頭檔 (如，math.h) 編寫所需呼叫程式。編譯時也必須將 Client Stab 和 XDR 檔案加入連結，範例如下：

```
$ cc –o math_req math_req.c math_clnt.c math_xdr.c  –lns1
```

(5) 遠端程序執行：在遠端電腦上執行遠端程序，一般都採用幕後執行模式(&)，範例如下：

（163.15.2.62）

$ math_proc　&

(6) 客戶端程序呼叫：在客戶端（163.15.2.30）呼叫遠端程序，譬如，遠端主機位址為163.15.2.62（linux-1），程序有兩個傳遞參數：20、50，範例如下：

$ math_req 163.15.2.62　20　50　　或

$math_req linux-1　20　50

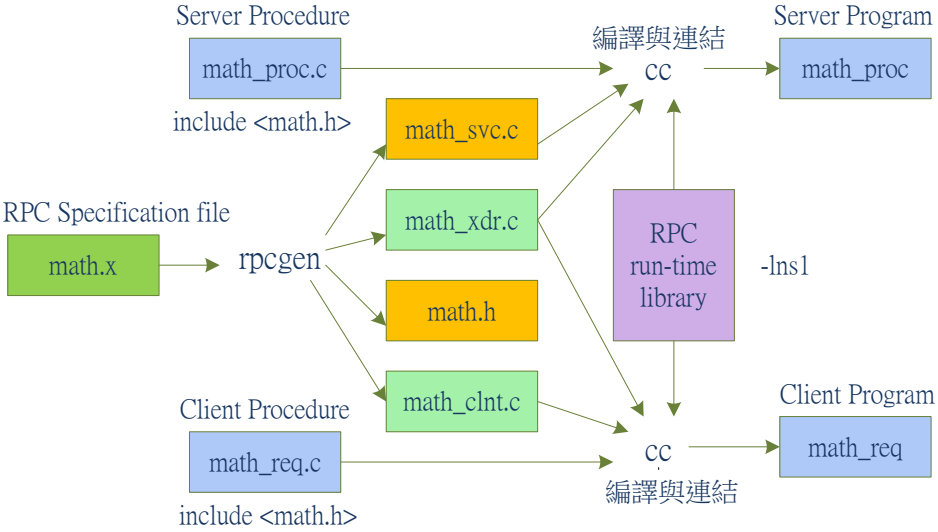

圖 9-6 RPC 程式開發之功能圖

9-5 RPC 程式範例

首先我們建立一個簡單遠端程序範例，來探討它的開發過程，也許會讓讀者比較容易進入情況。我們構想是伺服端提供三個數學運算的遠端程序讓客戶端呼叫，伺服端收到客戶端所傳的參數，經過計算後再傳回結果給客戶端。三個遠端程序功能如下：

- **add()**：客戶端呼叫該程序時，會攜帶兩個整數（Integer）參數給遠端程序，遠端程序將兩個參數相加後，傳回給客戶端。
- **multiply()**：客戶端呼叫該遠端程式時，會攜帶兩個整數參數，執行後傳回兩數值的乘績。
- **cube()**：客戶端傳遞一個整數參數，遠端程序傳回該參數三次方的值。

我們在伺服端製作上述的運算程式，當客戶端以 RPC 呼叫執行，伺服端運算後傳回結果給客戶端。我們依照上一節所敘述的步驟來建構如下：

9-5-1 建立規格檔 - math.x

```
/*
 *    math.x
 */
/* data structure used by procedure */
struct intpair {
        int a;
        int b;
};
/* procedure definition */
program MATHPROG {
        version MATHVERS {
                int ADD(intpair)    =1;
                int MULTIPLY(intpair)=2;
                int CUBE(int)              =3;
        } = 1;
} = 0x20000001;
```

規格檔的副檔名必須是『.x』，它包含程序呼叫之間的參數宣告和遠端程序名稱。範例中包含如下：

★ **傳遞參數宣告：**結構參數 intpair 中包含兩個整數 a 和 b，此參數作為客戶端和伺服端傳遞資料使用。在此宣告後經由 rpcgen 編譯後，會產生 XDR 的資料轉換檔。

★ **程式名稱：**以『**program**』關鍵字宣告程序名稱。如上例為 MATHPROG { ..... } = 0x20000001，也表示此程式號碼為 0x20000001，大掛號內（{ ... }）為程式內容。

★ **版本名稱：**以『**version**』關鍵字宣告版本名稱。如上例為 MATHVERS { ... } = 1，也表示此版本號碼是 1，大掛號內（{ ... }）包含著遠端程序的函數名稱。

★ **程序名稱：**宣告程序名稱。如上例中 int ADD(intpair) = 1，int 表示該程序傳回一個整數，所接收參數是 intpair，而該程序編號為 1。

由上例可以瞭解，規格檔只要制定程序號碼、版本、以及程序名稱，並宣告所傳遞的參數，另外，這些規格也是給伺服器和客戶端編寫程式的依據。

9-5-2 編譯規格檔 - rpcgen

利用 rpcgen 編譯規則檔（math.x）如下：

```
$ rpcgen   math.x
```

編譯後會產生下列檔案：

- **math.h**：程式標頭檔，作為編寫遠端程序和客戶端呼叫程序的依據。
- **math_clnt.c**：客戶基礎檔（Client Stub），一般都希望使用者不要去修改它。
- **math_svc.c**：伺服基礎檔（Server Stub），一般也希望使用者不要修改它。
- **math_xdr.c**：XDR 資料格式轉換檔，不要修改它。

以下分別介紹這些檔案：

(A) 標頭檔 – math.h

經過 rpcgen 編譯後所產生的 math.h 內容如下：

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#ifndef _MATH_H_RPCGEN
#define _MATH_H_RPCGEN

#include <rpc/rpc.h>

#ifdef __cplusplus
extern "C" {
#endif

struct intpair {
        int a;
        int b;
```

```
};
typedef struct intpair intpair;

#define MATHPROG 0x20000001
#define MATHVERS 1

#if defined(__STDC__) || defined(__cplusplus)
#define ADD 1
extern    int * add_1(intpair *, CLIENT *);
extern    int * add_1_svc(intpair *, struct svc_req *);
#define MULTIPLY 2
extern    int * multiply_1(intpair *, CLIENT *);
extern    int * multiply_1_svc(intpair *, struct svc_req *);
#define CUBE 3
extern    int * cube_1(int *, CLIENT *);
extern    int * cube_1_svc(int *, struct svc_req *);
extern int mathprog_1_freeresult (SVCXPRT *, xdrproc_t, caddr_t);

#else /* K&R C */
#define ADD 1
extern    int * add_1();
extern    int * add_1_svc();
#define MULTIPLY 2
extern    int * multiply_1();
extern    int * multiply_1_svc();
#define CUBE 3
extern    int * cube_1();
extern    int * cube_1_svc();
extern int mathprog_1_freeresult ();
#endif /* K&R C */

/* the xdr functions */

#if defined(__STDC__) || defined(__cplusplus)
```

```
        extern    bool_t xdr_intpair (XDR *, intpair*);

        #else /* K&R C */
        extern bool_t xdr_intpair ();

        #endif /* K&R C */

        #ifdef __cplusplus
}
```

基本上，標頭檔不希望被使用者修改，只要當編寫遠端程式和客戶端程式時，將它加入包含檔（Include File）即可，但我們必須了解其內容，才可依照標頭檔規範來編寫程式。

首先在標頭檔裡宣告傳遞參數 intpair，以及程式號碼(MATHPROG)和版本號碼(MATHVERS)。並且宣告三對程序結構，其中 add_1() 為客戶端呼叫 add() 程序，而 add_1_svc 為伺服端的遠端程序，兩者成為一對程序；相同的，multply_1() 與 multiply_1_svc()、cube_1() 與 cub_1_svc() 也各自成一對呼叫與被呼叫的程序。每一程序加入 _1，這表示版本 1 的意思。當我們在編寫遠端程序或客戶端程式都必須依照這些宣告內容來製作，雙方程式才會一致性。

(B) 客戶基礎檔 - math_clnt.c

math_cln.c 是 math.x 經過 rpcgen 編譯後所產生的，基本上也是不希望使用者去編改它。客戶基礎檔（Client Stub）主要是作為呼叫程式和遠端被呼叫程式之間的交換檔，因此，針對程序之間的交換格式必需給予標準規範，如 ADD() 函數以 add_1(intpair *argp, CLIENT *clnt) 作為呼叫格式。在每一個呼叫函數中都會包含較低層式的 clnt_call() 函數和有關 XDR 轉換函數。

math_cln.c 程式範例如下：

```
# cat math_cln.c
/*
* Please do not edit this file.
* It was generated using rpcgen.
*/

#include <memory.h> /* for memset */
#include "math.h"
```

```
/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = { 25, 0 };

int *
add_1(intpair *argp, CLIENT *clnt)
{
        static int clnt_res;

        memset((char *)&clnt_res, 0, sizeof(clnt_res));
        if (clnt_call (clnt, ADD,
        (xdrproc_t) xdr_intpair, (caddr_t) argp,
        (xdrproc_t) xdr_int, (caddr_t) &clnt_res,
        TIMEOUT) != RPC_SUCCESS) {
        return (NULL);
      }
      return (&clnt_res);
}
int *
multiply_1(intpair *argp, CLIENT *clnt)
{
      static int clnt_res;

      memset((char *)&clnt_res, 0, sizeof(clnt_res));
      if (clnt_call (clnt, MULTIPLY, (xdrproc_t) xdr_intpair, (caddr_t) argp,
                  (xdrproc_t) xdr_int, (caddr_t) &clnt_res,
                  TIMEOUT) != RPC_SUCCESS) {
                        return (NULL);
        }
        return (&clnt_res);
}
int *
cube_1(int *argp, CLIENT *clnt)
{
      static int clnt_res;

      memset((char *)&clnt_res, 0, sizeof(clnt_res));
        if (clnt_call (clnt, CUBE, (xdrproc_t) xdr_int, (caddr_t) argp,
                  (xdrproc_t) xdr_int, (caddr_t) &clnt_res,
```

```
                TIMEOUT) != RPC_SUCCESS) {
                        return (NULL);
        }
        return (&clnt_res);
}
int *
cube_1(int *argp, CLIENT *clnt)
{
        static int clnt_res;

        memset((char *)&clnt_res, 0, sizeof(clnt_res));
        if (clnt_call (clnt, CUBE, (xdrproc_t) xdr_int, (caddr_t) argp,
                (xdrproc_t) xdr_int, (caddr_t) &clnt_res,
                TIMEOUT) != RPC_SUCCESS) {
                        return (NULL);
        }
        return (&clnt_res);

}
```

(C) 伺服基礎檔 - math_svc.c

伺服基礎檔（Server Stub）也是規格檔（math.x）經由 rpcgen 編譯所產生，基本上也是不希望使用者去編改它。以下是 math_svc.c 的檔案範例，我們可以發現伺服程式的主程式（main()）是在基礎檔內，因此爾後編寫遠端程式時，就不需要再編寫主程式，而只要針對遠端程序編寫即可。另外，伺服基礎檔也是 Client/Server 程序之間呼叫的交換程序，因此也必須制定遠端程序的標準規範，以及有關 XDR 資料轉換的程序。math_svc.c 的程式範例如下：

```
# cat math_svc.c
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include "math.h"
#include <stdio.h>
#include <stdlib.h>
#include <rpc/pmap_clnt.h>
#include <string.h>
#include <memory.h>
#include <sys/socket.h>
#include <netinet/in.h>

#ifndef SIG_PF
#define SIG_PF void(*)(int)
#endif

static void

mathprog_1(struct svc_req *rqstp, register SVCXPRT *transp)
```

```
{
        union {
                intpair add_1_arg;
                intpair multiply_1_arg;
                int cube_1_arg;
                } argument;
        char *result;
        xdrproc_t _xdr_argument, _xdr_result;
        char *(*local)(char *, struct svc_req *);

        switch (rqstp->rq_proc) {
                case NULLPROC:
                        (void) svc_sendreply (transp, (xdrproc_t) xdr_void, (char *)NULL);
                        return;
                case ADD:
                        _xdr_argument = (xdrproc_t) xdr_intpair;
                        _xdr_result = (xdrproc_t) xdr_int;
                        local = (char *(*)(char *, struct svc_req *)) add_1_svc;
                        break;
                case MULTIPLY:
                        _xdr_argument = (xdrproc_t) xdr_intpair;
                        _xdr_result = (xdrproc_t) xdr_int;
                        local = (char *(*)(char *, struct svc_req *)) multiply_1_svc;
                        break;
                case CUBE:
                        _xdr_argument = (xdrproc_t) xdr_int;
                        _xdr_result = (xdrproc_t) xdr_int;
                        local = (char *(*)(char *, struct svc_req *)) cube_1_svc;
                        break;
                default:
                        svcerr_noproc (transp);
                        return;
        }
        memset ((char *)&argument, 0, sizeof (argument));
        if (!svc_getargs (transp, _xdr_argument, (caddr_t) &argument)) {
                svcerr_decode (transp);
                return;
        }
        result = (*local)((char *)&argument, rqstp);
        if (result != NULL && !svc_sendreply(transp, _xdr_result, result)) {
                svcerr_systemerr (transp);
        }
```

```
        if (!svc_freeargs (transp, _xdr_argument, (caddr_t) &argument)) {
                fprintf (stderr, "unable to free arguments");
                exit (1);
        }
        return;
}
int main (int argc, char **argv)
{
        register SVCXPRT *transp;
        pmap_unset (MATHPROG, MATHVERS);
        transp = svcudp_create(RPC_ANYSOCK);
        if (transp == NULL) {
                fprintf (stderr, "cannot create udp service.");
                exit(1);
        }
        if (!svc_register(transp, MATHPROG, MATHVERS, mathprog_1, IPPROTO_UDP)) {
                fprintf (stderr, "unable to register (MATHPROG, MATHVERS, udp).");
                exit(1);
        }
        transp = svctcp_create(RPC_ANYSOCK, 0, 0);
        if (transp == NULL) {
                fprintf (stderr, "cannot create tcp service.");
                exit(1);
        }
        if (!svc_register(transp, MATHPROG, MATHVERS, mathprog_1, IPPROTO_TCP))
{
        fprintf (stderr, "unable to register (MATHPROG, MATHVERS, tcp).");
        exit(1);
        }
        svc_run ();
        fprintf (stderr, "svc_run returned");
        exit (1);
                /* NOTREACHED */

}
```

(D) 資料格式轉換檔 – math_xdr.c

同樣的，資料格式轉換檔也不希望使用者去修改它，它主要是做傳遞參數資料結構的轉換，對我們編寫程式較沒有影響。但如果使用者希望作較低層次的程式編寫，就必須參考有關 XDR 庫存函數的使用方法，這不在本書的介紹範圍內。

```
# cat math_xdr.c
/*
   * Please do not edit this file.
   * It was generated using rpcgen.
*/
#include "math.h"
bool_t
xdr_intpair (XDR *xdrs, intpair *objp)
{
        register int32_t *buf;
```

```
    if (!xdr_int (xdrs, &objp->a))
                return FALSE;
    if (!xdr_int (xdrs, &objp->b))
                return FALSE;
    return TRUE;

}
```

9-5-3 編寫遠端程序 – math_proc.c

遠端程序放置於伺服器端，可在伺服端製作，也可以在客戶端完成後再檔案傳輸到伺服器端。由 math_svc.c 檔案中，可以發現主程式已在該檔案內規劃完成，我們只要編寫欲被呼叫的程序即可，當然編寫時必須參考標頭檔 (math.h) 和遠端基礎檔 (math_svc.c) 內針對資料和函數的宣告。math_proc.c 程式範例如下：

```
#include <rpc/rpc.h>
/* math.h is generated by rpcgen */
#include "math.h"

int * add_1_svc(pair, sv)
intpair *pair;
struct svc_req *sv;
{
    static int result;
    result = pair->a + pair->b;
    return(&result);
}

int * multiply_1_svc(pair, sv)
intpair *pair;
struct svc_req *sv;
{
    static int result;
    result = pair->a * pair->b;
    return(&result);
}

int * cube_1_svc(base, sv)
int *base;
struct svc_req *sv;
{
    static int result;
    int baseval = *base;
```

```
    result = baseval * baseval * baseval;
    return(&result);
}
```

其中 add_1_svc()、multiply_1_svc() 與 cube_1_svc 在標頭檔（math.h）已宣告完成，也表示版本為 1（_1），另外結構資料 svc_req 也是在標頭檔宣告。其它有關函數功能的實現，也必須在遠端程序內製作完成，譬如，本範例的乘法、加法和三次方的運算。

9-5-4 編寫客戶端程式 – math_req.c

其實編寫客戶端程式比伺服端程序較為複雜，其中會牽涉到一些 RPC 庫存函數的功能呼叫，當然伺服端也會有功能呼叫，但大多實現在基礎檔（Server Stub）內，使用者也大多不需要去修改它。但客戶端也許會針對不同伺服端要求程序呼叫，因此有關功能呼叫部份必需由使用者自行規劃。同樣的，編寫時也必須參考標頭檔（math.h）和基礎檔（math_cln.c）內有關資料和程序的宣告。首先我們來看 math_req.c 的客戶端程式範例，在下一節再來探討 RPC 程式庫的呼叫方法，math_req.c 範例如下：

```
# cat math_req.c
/*
 * Client Program, math_req.c
 */
#include <stdio.h>
#include <rpc/rpc.h>
#include "math.h"

main (argc, argv)
int argc;
char *argv[];
{
    CLIENT *cl;
    intpair numbers;
    int *result;

    if (argc != 4) {
        fprintf(stderr, "%s: usage: %s server num1 num2\n",
                argv[0], argv[0]);
        exit(1);
    }
    cl = clnt_create(argv[1], MATHPROG, MATHVERS, "tcp");
```

```
    if (cl == NULL) {
        clnt_pcreateerror(argv[1]);
        exit(1);
    }
    numbers.a = atoi(argv[2]);
    numbers.b = atoi(argv[3]);

    result = add_1(&numbers, cl);
    if (result == NULL) {
        clnt_perror(cl, "add_1");
        exit(1);
    }
    printf("The add (%d,    %d) procedure returned %d\n",
            numbers.a, numbers.b, *result);

    result = multiply_1(&numbers, cl);
    if (result == NULL) {
        clnt_perror(cl, "multiply_1");
        exit(1);
    }
    printf("The multiply(%d,    %d) procedure returned    %d\n",
            numbers.a, numbers.b, *result);

    result = cube_1(&numbers.a, cl);
    if (result == NULL) {
        clnt_perror(cl, "cube_1");
        exit(1);
    }
    printf("The cube (%d) procedure returned %d\n",
            numbers.a, *result);
    exit(0);
}
```

在上例中使用一個特殊函數 clnt_creat()，此為客戶端連接遠端程序的功能呼叫，我們將在下一節介紹。

9-5-5 編譯程式與執行 - cc

● **伺服器端編譯如下：**(lnsl 中的 l 是 L 的小寫)

```
$ cc –o math_proc math_proc.c math_svc.c math_xdr.c –lnsl
```

● **客戶端編譯如下：**

```
$ cc –o math_req math_req.c math_clnt.c math_xdr.c –lnsl
```

● **伺服器端執行如下：**(主機位址 163.15.2.62)

```
$ math_proc    &
```

如欲刪除伺服程式，則利用 ps –ef 查詢出該程式的 PID 號碼，再利用 kill 命令將其刪除。

● **客戶端執行如下：**

```
$ math_req 163.15.2.62    45    21
```

執行結果如下：

```
$ math_req 163.15.2.62 45 21
The add (45,    21) procedure returned 66
The multiply(45,    21) procedure returned    945
The cube (45) procedure returned 91125
```

9-5-6 RPC 訊息報告 – rpcinfo

我們可以利用 rpcinfo 命令來查詢 RPC 的使用情況，命令格式如下：(root 系統管理者才有權限使用)

```
# rpcinfo –p [host]
# rpcinfo [-n portnum] –u host program [version]
# rpcinfo [-n portnum] –t host program [version]
# rpcinfo –b program version
# rpcinfo –d program version
```

如查詢本機電腦上的 RPC 訊息如下；

```
# rpcinfo –p
   program     vers    proto     port
    100000        2      tcp      111   portmapper
    100000        2      udp      111   portmapper
    100021        1      udp     1024   nlockmgr
    100021        3      udp     1024   nlockmgr
    100021        1      tcp     1024   nlockmgr
    100021        3      tcp     1024   nlockmgr
    100024        1      udp     1013   status
    100024        1      tcp     1015   status
```

```
536870920       1       udp       1027
536870920       1       tcp       1026
536870913       1       udp       1028

        536870913       1       tcp       1027
```

9-6 RPC 庫存函數

雖然規格檔經過 rpcgen 編譯後，會自動產生許多相關的 RPC 功能呼叫，但在許多情況下也需要使用者自行規劃使用。在 Unix/Linux 系統下，RPC 庫存函數（RPC Library）包含很多，一般而言都屬於比較低階（Low-level）的功能呼叫，我們只列出一些較常用的，如需要更詳細說明，請參考有關 RPC 的使用手冊（本書附錄的參考資料裡有列出）。

9-6-1 伺服端庫存函數

依照我們的範例，伺服端的 RPC 功能呼叫都存放於基礎檔（math_svc.c）內，一般較常用的有下列兩種：

(A) svctcp_create()

此功能呼叫是希望核心系統開啟一個 RPC 傳輸連線，而此連線是建立在 TCP 傳輸點（Transport）上，又此 TCP 傳輸點是建立在 Socket 通訊端點上，程序語法如下：

```
SVCXPRT *svc
svc = svctcp_create(socket, sendsz, recvsz)
    int socket;

    u_int sendsz, recvsz;
```

由 svctcp_create() 呼叫所產生的 RPC 傳輸點（SVCXPRT）也稱之為『TCP-based RPC』，它是屬於緩衝器型的輸出/輸入裝置（Buffer I/O），因此必須定義接收緩衝器（recvsz）的大小，和傳送緩衝器的空間大小（sendsz），而該通訊端點是建立在已連接成功的 Socket 識別碼（socket ID）上。

(B) svcudp_create()

此功能呼叫也如同 svctcp_create() 一樣，但它是產生一個 UDP/IP-based 的傳輸點上，svcudp_create() 語法如下：

```
SVCXPTR *svc
svc = svcudp_create(socket)

    int socket;
```

此功能呼叫為非連接方式，因此不需要定義緩衝器大小。又 UDP 並未事先建立 Socket 的通訊端點，因此 socket 的內涵可定義為 RPC_ANY_SOCK，表示任意所產生的通訊端點。如果呼叫回應為 NULL，表示開啟失敗。

9-6-2 客戶端庫存函數

客戶端的功能呼叫不一定全然存放於基礎檔內，許多地方需要使用者在應用程式上編寫，因此，我們就必須特別去瞭解它。我們可以發現大部份的功能呼叫，都是針對一個特殊資料結構作處理，此資料結構為 CLIENT，表示 Client/Server 之間的通訊端點。以下介紹較常用的功能呼叫：

(A) clnt_create()

clnt_create() 為客戶端最主要的功能呼叫，主要功能是產生一個 Client/Server 的通訊連線，呼叫語法如下：

```
CLENT *clnt
clnt = clnt_create(host, prognum, versnum, protocol)
      chat *host
      u_long prognum, versnum;

char *protocol
```

參數中，host 表示連線主機；prognum 為程式號碼（在規格檔中宣告）；versnum 是程式版本（也在規格檔中宣告）；protocol 表示連線採用何種通訊協定；呼叫成功會傳回一個連線號碼，否則傳回空值（clnt = NULL）。由這個呼叫，我們大略可以瞭解它的連線方式，其表示用某一種通訊協定連結到遠端主機上，呼叫某一程式號碼下的一個程式版本。

參數中 protocol 表示連接方式，不同的網路環境有各自的連接方式，這牽涉到網路系統環境的安裝與管理，一般有下列幾種：

(1) "natpath"：使用 Network Selection 通訊服務。clnt_create() 依照環境變數 NETPATH 的參數值連結，以第一個連結成功為主，其可能是連接導向或非連接方式。

(2) "circuit_n"：使用 Network Selection 通訊服務，也是依照環境變數 NETPATH 中連接導向的通訊方式。

(3) "datagram_n"：同上，但採用非連接方式。

(4) "visible"：使用 Network Selection 通訊服務。clnt_create() 依照 /etc/netconfig 所設定的連接方式，但以第一個連接成功為主，可能是連接導向或非連接方式。

(5) "circuit_v"：同上，但採用連接導向的通訊方式。

(6) "datagram_v"：同上，但採用非連接方式。

(7) "tcp"：使用 TCP 協定，亦是一般 Socket 系統呼叫連結方式。

(8) "udp"：使用 UDP 協定，下層也是透過 Socket 系統呼叫連結。

如果採用 Network Selection 服務，在客戶端和伺服器上的主機電腦，必須安裝 Network Selection 通訊軟體。Network Selection 服務是將一般網路介面以變數名稱來代表，讓程式設計者存取網路資源，就如同讀取變數內容一樣方便。目前也有許多系統安裝 Network Selection，這讓讀者自行去探討，作者不再贅言。

在我們範例（math_req.c）中，clnt_create() 格式如下：

cl = clnt_create("linux-1", 0x20000001, 1, "tcp")

表示客戶端對伺服器端主機（linux-1 或 IP 位址），開啟 TCP 連線（也是 Socket Based），並呼叫程式號為 0x20000001 的遠端呼叫，而呼叫的程序版本為 1 內的遠端程序。程式號碼和版本號碼皆在標頭檔（math.x）內宣告。如果呼叫成功，便會傳回一個客戶端指標位址，如呼叫失敗便傳回空值（cl = NULL）。

(B) clnt_destroy()

此功能呼叫為關閉已開啟的 RPC 通訊連線，如果該連線是 TCP Based，同時會自動關閉相關的 Socket 端點。clnt_destroy() 的語法如下：

```
clnt_destroy(clnt)
CLIENT *clnt;
```

(C) clnt_control()

此功能呼叫是針對已開啟成功的 RPC 連線作功能性設定，其中包含：溢時時間（time out）、UDP 重新傳送的溢時時間、雙方 Socket 的連線端點、以及其它狀態。clnt_control() 的語法如下：

```
bool_t bol;
bol = clnt_control(clnt, request, info)
      CLIENT *clnt;
      int request;
char *info;
```

其中參數 clnt 為欲控制的 RPC 連線；request 為控制項目；如果控制參數中需要傳送訊息，便存放在 info 變數內。如果呼叫成功便會回應 TRUE；否則回應 FALSE。

一般控制項目（request）有下列幾種：

(1) CLSET_TIMEOUT：設定整體的溢時時間（total time-out）。

(2) CLGET_TIMEOUT：讀取已設定的整體溢時時間。

(3) CLGET_FD：取得該 RPC 連線的 Socket 連線號碼。

(4) CLSET_FD_CLOSE：設定當 clnt_destroy 執行時，同時關閉 Socket 連線。

(5) CLSET_FD_NCLOSE：設定當 clnt_destroy 執行時，不關閉 Socket 連線。

(6) CLGET_SERVER_ADDR：取得該 RPC 連線的伺服端 IP 位址。

另外針對 UDP 連線有下列兩個控制：

(1) CLSET_RETRY_TIMEOUT：設定重試的溢時時間。

(2) CLGET_RETRY_TIMEOUT：取得重試的溢時時間。

(D) clnt_call()

當 RPC 連線成功（clnt_create()）後，再利用 clnt_call() 來呼叫遠端程序，一般 clnt_call() 函數都存放在客戶基礎檔（math_clnt.c）內，作為客戶端程式呼叫遠端程式之間的轉換，大多不希望使用者更改。我們由它的呼叫程序可大略瞭解 RPC 的運作方式，clnt_call() 語法如下：

```
enum clnt_stat
clnt_call(clnt, procnum, inproc, in, outproc, out, timeout)
        CLIENT *clnt;
        u_long procnum;
        xdrproc_t inproc, outproc;
        char *in, *out;
        struct timeval timeout;

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

};
```

我們用 math_clnt.c 範例中的：

『clnt_call (clnt, ADD, (xdrproc_t) xdr_intpair, (caddr_t) argp, (xdrproc_t) xdr_int, (caddr_t) &clnt_res, TIMEOUT)』

來說明 clnt_call() 中的參數值，其中參數 clnt 為 RPC 連線識別值；procnum 為遠端程序號碼（ADD）；inproc 為傳遞給遠端伺服端的 XDR 轉換函數（(xdrproc_t) xdr_intpair）；而 in 為傳遞參數((caddr_t) argp); 相同的，outproc 是由遠端傳回來參數的 XDR 轉換函數((xdrproc_t) xdr_int)；out 參數為遠端傳回之數值。

當執行 clnt_call() 後，會等待到系統回應才會返回(Blocking)，如返回後結果 RPC_SUCCESS 表示執行成功，否則會回應其它不成功的原因。如果系統沒有回應，則參數 timeout 表示溢時時間。

除了以上功能呼叫外，尚有下列三個較常用的函數，分別說明如下：

- **clnt_pcreateerror()**：當呼叫 clnt_create() 失敗時，此函數會將呼叫失敗原因的訊息，寫入標準錯誤輸出地方。
- **clnt_perror()**：執行介面呼叫失敗時，呼叫此函數，它會將失敗的原因表示在標準錯誤輸出。
- **clnt_freeres()**：此函數的功能是釋放程式介面執行時，所取用之記憶體。當呼叫介面程式後，執行此功能呼叫來釋放記憶體。

9-7 RPC 定義語言

RPC**『定義語言』(Definition Language)** 是製作 RPC 規格檔的描述語言，其中包含規格檔的製作方法和抽象資料的表示法，以下分別介紹之。

9-7-1 規格檔之製作

製作**『規格檔』(Specification File)**必需依照 RPC 定義語言的標準來敘述，才能被 rpcgen 所編譯。所編寫規格檔的副檔名，一定必須是**『.x』**。一個**『.x』**檔案裡必需包含交換資料的**『資料型態』(Data Type)**、**『程式號碼』(Program Number)**、**『程式版本』(Program Version)** 和**『程序名稱』(Procedure Name)**。它的定義非常簡單，基本表示法為一個程式名稱包含一組程式版本，又

一個程式版本包含一組遠端程序名稱，所包含的內容以左右大掛號（{ ... }）標示。以下列步驟來說明：

(A) 宣告程式號碼

以關鍵字 program 宣告一個程式號碼，其名稱為 identifier，而程式號碼是一個不含負數的整數 value。在一個程式號碼可包含若干個版本宣告，格式如下：

```
program identifier {
        version_1;
        ……..
        version_n
} = value;
```

(B) 宣告程式版本

利用關鍵字 version 宣告一個程式版本，版本名稱為 identifier，版本號碼是 value(identifier = value)。在一個程式版本中也可以包含若干個程序宣告，格式如下：

```
version identifier {
         procedure_1;
         ……
         procedure_n;
} = value;
```

(C)宣告程序

宣告程序和一般 C 語言的語法非常類似，必需宣告程序傳遞和傳回參數的資料型態(如，int、unsigned int、void、char)。procedure_name 為程序名稱，value 是程序號碼。

```
data_type procedure_name( data_type) = value;
```

(D) 範例說明

規格檔範例如下：

```
program TIMEINFO {
        version TIMEVERS {
                unsigned int GETTIME (void) = 1;
          } = 1;
} = 0x20000006;
```

以上範例定義一個名稱為 TIMEINFO 的程式，它的程式號碼為 0x20000006。該程式中包含一個版本名稱，其版本名稱為 TIMEVERS，版本號碼是 1。此程式版本也只包含一個遠端程序（GETTIME），程序號碼為 1。呼叫 GETTIME 程序時，並沒有傳遞參數（void），但傳回一個沒有符號的整數(unsigned int)。此範例經過 rpcgen 編譯後，所產生的標頭檔會包含下列宣告值 :(部分檔案) :

```
# define TIMEINFO ((u_long) 0x20000006)
# define TIMEVERS ((u_long) 1)
# define GETTIME ((u_long) 1)

extern u_int *gettime_1();
```

9-7-2 資料型態之宣告

RPC 定義語言的資料宣告也非常類似 ANSI C 語言的宣告方式，但為了符合交換的抽象資料型態，RPC 另外自行定義宣告方式。定義語言所宣告的資料型態經由 rpcgen 編譯後，也必須符合 ANSI C 的資料型態。以下分別介紹 RPC 所宣告的資料型態。

(A) 常數（Consttant）

RPC 定義常數的格式如下：

```
const identifier = integer_value;
```

以關鍵字 const 宣告常數，其名稱為 identifier，而值是 integer_value。範例如下：

```
const MAX_COUNTER = 1024;
```

經過 rpcgen 編譯後，結果為：

```
# define MAX_COUNTER 1024
```

(B) 結構（Structures）

RPC 的結構宣告與 ANSI C 的型態相同，範例如下：

```
struct intpair {
        int a;
        int b;
};
```

經過 rpcgen 編譯後，結果為：

```
struct intpair {
    int a;
    int b;
};
typedef struct intpair intpair;
```

關鍵字 typedef 是讓 intpair 來取代 struct intpair，因此在 RPC 程式中只要用 intpair 宣告即可。

(C) 列舉（Enumerations）

如同結構一樣，RPC 宣告列舉也和 ANSI C 語言相同，範例如下：

```
enum light {
    RED = 0,
    AMBER = 1,
    GREEN = 2
};
```

經過 rpcgen 編譯後，結果為：

```
enum light {
    RED = 0,
    AMBER = 1,
    GREEN = 2
};
typedef enum light light;
```

(D) 聯集（Unions）

RPC 的聯集定義有點類似 C 語言，但是兩者在本質上有相當的差異。C 語言的聯集是定義一串元件的儲存空間，而 RPC 的聯集是條件式的資料型態規格。我們用一個範例來說明 RPC 聯集的使用，此範例為某一個程序接收到訊息的處理情形：

(1) 程序接收到一個字元陣列（Array），其中包含著以參數描述的使用者名稱。

(2) 判斷該使用者是否已登入系統，如果使用者已登入，則將原字元陣列填入使用者登入時間（字元），並回應給傳送者。如果使用者並未登入，則回應空白資料（NULL）。如果根本沒有這位使用者，則回應錯誤訊息（整數表示）。

在這個範例中，該程序可能回應三種資料型態的一種：字元陣列、整數或空白（NULL 或 void）。RPC 聯集定義此類型資料的轉送，它包含選擇（switch）來決定何時要傳送何種資料型態。RPC 聯集的語法如下：

```
union identifier switch (declaration) {
        case value_1 : declaration_2;
        case value_2 : declaration_2;
        .....
        default : declaration_n;
}
```

聯集變數名稱為 identifier，而條件判斷之變數是 declaration，在每一相對條件都有一個專屬宣告。

如果依照前述的範例，則 RPC 聯集的宣告如下：

```
const MAX_BUF = 30;
union time_results switch (int status) {
    case 0:    char timeval[MAX_BUF];
    case 1:    void;
    case 2:    int reason;
};
```

這表示 time_result 可能有三種資料型態：字元陣列、空值（void）或整數，到底會使用何種型態，則依照 status 變數的內容而定。此例經由 rpcgen 編譯後，結果如下：

```
# define MAX_CHAR 30
struct time_results {
    int status;
    union {
        char timeval[MAX_CHAR];
        int reason;
    } time_result_u;
```

```
};

typedef struct time_results time_results;
```

在 rpcgen 編譯後，好像和原來 RPC 規格檔有很大的差異，此點必須特別說明如下：

(1) 結構資料所包含的聯集資料之間，如有相同的變數名稱，則在聯集的變數名稱的後面加入『**_u**』。

(2) void 宣告被省略掉，因它只在沒有訊息傳送的情況下才會發生。

(3) 結構變數之名稱和原來 RPC 的聯集變數名稱一樣。

(4) 當實現程序時，必需確定 status 變數是否只有 0、1 和 2 三種數值的機會，如以 RPC 規格-檔裡所描述，確實只能這三種數值。因此在程序中必需依照結構變數特殊處理：

- 如果放置 0 到 status 變數內，則必須將資料填入 time_results_u.timeval 陣列中。
- 如果設定 status 為 1 時，則不需填入任何值到聯集內。
- 如果設定 status 為 2 時，則必須填入一個整數到 time_results_u.reason 元件內。

(5) 當客戶端收到程序的回應時，首先必須測試 status 的內容，再來決定應接收何種型態的資料，如此便不會發生錯誤。

(E) 型態定義（Type Definitions）

RPC 的型態定義和 C 語言完全相同，rpcgen 編譯後也沒有改變，範例如下：

```
typedef long conuter_t;
```

(F) 陣列宣告（Declarations of Arrays）

RPC 定義語言允許使用宣告含有結構（Structure）、聯集（Union）和型態定義（typedef）的陣列，同時也允許宣告固定長度和可變長度的陣列。對於固定長度的宣告較沒有問題，也和 C 語言相同，大部分 rpcgen 也不會作任何改變。但 C 語言並不提供可變長度陣列的宣告，因此，rpcgen 會做較特殊的處理。宣告固定長度和不固定長的陣列情況如下：

- 如宣告固定長度陣列，範例如下：

```
int proc_times [100];
```

表示整數陣列長度固定為 100，經過 rpcgen 編譯後也沒有改變。

- 如宣告可變長度陣列，範例如下：

```
typedef long x_records <50>;
```

這表示 x_records 陣列的長度可以由 0 到 50，資料型態是長整數。如經過 rpcgen 編譯後，結果如下：

```
typedef struct {
    u_int x_records_len;
    long *x_records_val;
} x_records;
```

因為 C 語言沒有可變長度陣列的宣告，因此必須用結構宣告來取代。結構中 x_records_len 代表第幾個陣列變數；而 x_records_val 是陣列內容的指標位址。原來規格檔內有最長限制 50，編譯後成為沒有最長限制。

(G) 指標宣告（Declarations of Pointers）

RPC 的指標宣告和 C 語言相同，rpcgen 也不會做任何改變，範例如下：

```
int    *nextp;
```

雖然指標宣告如同 C 語言，但使用在遠端程序呼叫中傳遞訊息，就和同一主機上傳遞有很大的不同點。如遠端程序呼叫使用指標參數時，而它所傳遞的是記憶體位址，在不同電腦之間的記憶體位址根本沒有意義。因此，必須考慮以下列方式宣告：

```
struct linked_list {
    int value;
    struct linked_list *nextp;
};
```

以上就是利用串列結構來表示，並可利用布林代數（value）來表示資料是否存在。

(H) 字串（Strings）

字串宣告可以是可變長度，範例如下：

```
string    name<50>;
```

rpcgen 編譯後，結果為：

```
char    *name;
```

雖然並沒有指定最長長度，但在程序編寫時儘量不要超過規格檔（.x）所限定的範圍。

(I) 布林值（Boolean Values）

以 RPC 定義語言宣告布林值範例如下：

```
bool    waiting;
```

rpcgen 編譯後結果為：

```
boot_t    waiting;
```

在 C 語言裡並沒有 boot_t 的資料型態，這必需利用 RPC 函數庫定義資料型態（typedef），使這個變數僅能填入 TRUE 或 FALSE。

(J) 空值（Void）

如同一般宣告，RPC 宣告一個空值的格式如下；

```
void;
```

但在 RPC 內宣告空值只有兩個地方：

(1) 聯集定義：在聯集元件中不傳回任何值。

(2) 程式定義：在程式宣告中不傳遞任何參數。

(K) 不明確資料（Opaque Data）

RPC 定義語言允許宣告不明確資料，表示資料型態並未指定，而當程式需要時在指定何種資料型態。固定長度的宣告語法為：

```
opaque    extra_bytes[1024];
```

rpcgen 編譯後為：

```
char    extra_bytes[1024];
```

可變長度的 RPC 宣告如下：

```
opaque    more_bytes<1024>;
```

rpcgen 編譯後，結果為：

```
struct {
    u_int    more_bytes_len;
    char    *more_bytes_val;
} more_bytes;
```

由此可見，rpcgen 將不明確資料以字元型態來表示。

9-8 XDR 資料表示協定

在執行遠端程序呼叫時，也許會跨越不同型態電腦來執行程序呼叫。不同型態的主機電腦或作業系統，對於**『抽象資料型態』(Abstract Data Type)**的表示法也許會有所不同，遠端程序呼叫之間所傳遞參數的表示數值，容易造成嚴重的錯誤，因此，在傳遞參數之前必需給予一種標準表示法，而接收端也以標準表示法來讀取參數，才能使傳遞訊息達到一致性。早期 Sun 公司就採用**『外部資料表示法』(eXternal Data Representation, XDR)**，將其崁入遠端程序呼叫內，來解決抽象資料表示不同的問題。

首先我們來探討不同電腦系統之間，抽象資料表示有可能發生哪些問題：

(1) 資料長度：一般宣告變數時都會給一個記憶體空間以供儲存，至於記憶體空間大小是由資料型態來決定。但對於不同電腦系統來說，針對某一資料型態給予的空間大小不一定相同。譬如，某一部電腦 (主機_A) 宣告一個整數給於 16 位元空間儲存，而其它電腦 (主機_B) 可能給予 32 位元空間。當主機_A 呼叫主機_B 上的程序，而攜帶一個整數參數 (16 位元)，主機_B 接收參數時便無法辨識該內容。

(2) 位元組順序：位元組的順序在不同電腦系統之間也許會不一樣，有的電腦最高位元在最右邊 (Intel CPU)，然而有的是在最左邊 (Motorola CPU)。如果之間沒有特殊轉換，那麼傳遞參數將會發生嚴重的錯誤。

(3) 資料表示：對於各種資料表示法也可能不同。譬如，帶符號的數值大小中，其數值是 2 的補數或 1 的補數；或浮點運算資料中，數值和指數大小的表示；或對於空值 (Null) 的表示，這些在不同電腦之間都有很大的差異。

(4) 序列（Alignment）：如果一個變數超過一個位元組，而必須使用兩個以上的位元組表示時，低位元組是在高或低記憶體位址，這在不同電腦之間表示也不一樣。譬如表示一個兩位元組的數值，Motorola CPU 以低記憶體位址放置較高的位元組；而 Intel CPU 則相反。

XDR 定義一種標準表示法，傳送端欲將資料傳送之前，先將其轉換成 XDR 表示法，接收端收到訊息時，再將資料轉換成自己的資料表示法，這樣雙方傳送資料就不會發生認定資料內容的錯誤。如圖 9-7 中，XDR 定義整數都以 32 位元長度表示(長整數或短整數)，而且使用 2 的補數、最高位元是最左邊的位元、最低位元是最右邊位元，而客戶端是以 16 位元表示短整數（int），長整數以 32 位元表示。伺服端也都以 32 位元表示長整數和短整數。當客戶端和伺服端之間資料格式，就利用 XDR 來轉換。

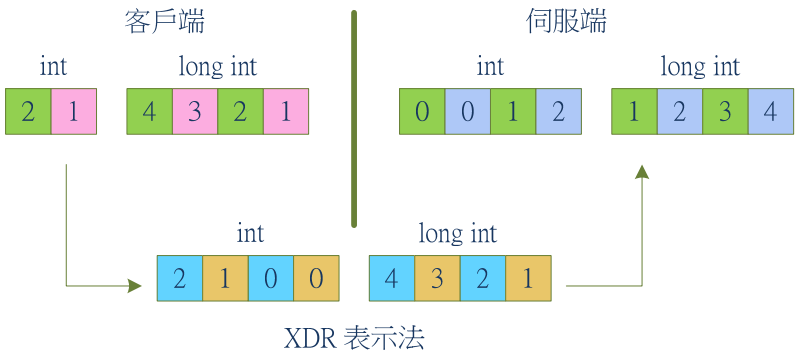

圖 9-7 XDR 表示範例

我們使用 rpcgen 編譯規格檔時，它會依照規格檔裡所宣告的傳遞參數，產生一個 XDR 資料格式轉換檔，副檔名為『**.xdr**』。編譯客戶端主程式或伺服端程序時，只要將該檔案連結進去即可。但如果編寫較低階的遠端程序(不使用 rpcgen)，則必須自行規劃 XDR 檔案，至於如何編寫 XDR 檔案，留給讀者自行探討或請參考其它書本。

9-9 RPC 的安全性

在一般 Internet 網路上應用遠端程序呼叫，我們希望每一個 RPC 呼叫請求或回應都能夠被驗證，以保證資料的安全性。試想客戶端利用一個 RPC 呼叫，它攜帶某一組參數請求，遠端程序利用這些參數來處理，並回應所需的數值，如果沒有經過適當的安全措施，將很容易使入侵者盜取資料或破壞系統。因此，Client/Server 之間的必需維護一種驗證訊息（Authentication Information）的

措施，伺服端可依照此驗證訊息來確定呼叫程序的使用者身分，而客戶端可依此訊息來認證資料確實是來自伺服端所傳送，而非偽裝資料。

一般 RPC 所使用的認證訊息可分為：**『憑據』(Credentials)** 和 **『證實器』(Verifier)** 兩部份。憑據是作為驗證使用者身分使用；而證實器是作為使用者傳送訊息的證實使用用，針對上述兩部份在不同的安全措施上，有不同的製作方法，稱之為 **『驗證措施』**，以下分別說明。(如圖 9-8 所示)

(1) None：在這種驗證措施之下，憑據和證實器都是空值，表示並沒有任何安全措施，如 9-8 (a) 所示。伺服端只要接收到任何呼叫請求，都給予處理並回應，相同的，客戶端收到任何回應都相信它是來自所呼叫的伺服端。這就如同我們的程式範例一樣，沒有處理任何安全措施。

(2) Unix：Unix 的驗證措施是來自 Sun NFS (Network File System) 的方法，也是 RPC 最早的應用環境。如 9-8 (b) 所示，憑據上填上有關的使用者訊息：主機名稱(Machine Name)、使用者 ID (User ID)、群組 ID (Group ID)，以及呼叫的時間戳記 (Time Stamp)。這表示遠端主機會驗證使用者身分，是否有足夠權限可以呼叫遠端程序，這應用在網路共享檔案上非常適合。由 9-8 (b)上可以看出，證實器上也是空值，表示不驗證資料傳遞的真實性。

(3) Secure：『安全性 RPC』(Secure RPC) 表示必需驗證使用者身分，以及防範入侵者偽裝盜取或破壞資料，其中包含 **『認證技術』** 和 **『加密技術』**，如 9-8 (c) 所示。一般 Secure RPC 都是透過客戶端和伺服端交換各自的私人金鑰，來產生秘密金鑰，以作為雙方通訊的憑據。

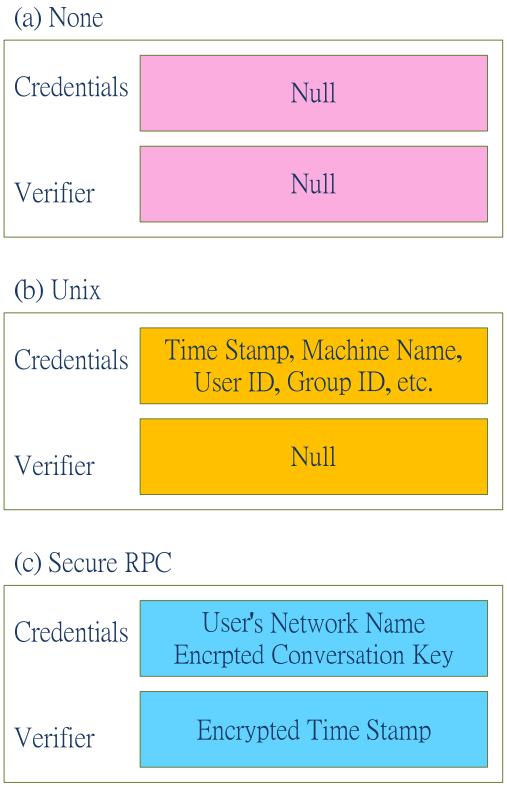

圖 9-8 RPC 之驗證措施

目前 Internet 網路上的電子商務應用非常普遍，甚至 RPC 程序呼叫也植入在各種應用系統之中，Secure RPC 的重要性更為顯著。如欲更進一步的學習 Secure RPC 製作方法，必須詳加研習有關較低階的 RPC 程序庫的使用方法，才能將一驗證技術植入 RPC 呼叫之中，這一方面請參考較詳細的 RPC 程式編寫技巧，在本書附錄中有些資料非常具有研讀的價值，另外有關驗證技術可以參考拙著**『Internet 網路安全與實務』**(近期出版)。

習題

1. 何謂『**遠端程序呼叫**』（**Remote Procedure Call, RPC**）？它和一般程序呼叫有何不同？

2. 請簡單敘述 RPC 的協定架構，並說明格協定層次所扮演的功能。

3. 請簡述說明 RPC 通訊連線採用 TCP 或 UDP 協定的考慮條件為何？

4. 請簡述說明 RPC 連線建立的運作方式。

5. 請簡述說明 RPC 程序呼叫的運作方式。

6. 何謂『**伺服基礎檔**』（**Server Stub**）？何謂『**客戶基礎檔**』（**Client Stub**）？其功能為何？

7. 請說明不同系統之間可能出現哪些資料不一致性的問題。

8. 請利用 RPC 製作一資料庫系統，伺服端維護一簡單的學生資料庫，包含學生姓名、學號、年齡、電話，並製作插入（Insert）、更新（Update）、查詢（Query）與刪除（Delete）的遠端程序，客戶端分別製作相對應的程序呼叫（inset_req、updat_req、query_req 與 delete_req），來呼叫遠端程序在資料庫上的管理動作。