Università degli Studi di Milano

Corso di Laurea in
Sicurezza dei Sistemi e delle Reti Informatiche.

Lezione 1 – RPC (Remote Procedure Calls)

ERNESTO DAMIANI

Sistemi di elaborazione dell'informazione

Modulo 4 - Unità Didattica 2

1.RPC (Remote Procedure Calls)

1.1 Descrizione di una RPC

RPC (Remote Procedure Calls) è una tecnica potente per costruire applicazioni distribuite basate sul paradigma client-server. RPC e si fonda sull'ampliamento della nozione convenzionale di chiamata di procedura locale e prevede che la procedura chiamata non debba essere nello stesso spazio d'indirizzamento della procedura chiamante. Anzi, chiamante e chiamato sono eseguiti in due processi diversi che possono trovarsi sullo stesso sistema o su sistemi diversi con una rete che li connette. Usando RPC, i programmatori di applicazioni distribuite evitano di dover conoscere dettagli dell'interfaccia con la rete. L'indipendenza del trasporto di RPC isola l'applicazione dagli elementi logici e fisici del meccanismo di comunicazione dei dati e consente all'applicazione di usare vari protocolli. RPC rende più facile programmare il modello di elaborazione client/server.

1.2 Il funzionamento di una RPC

Una chiamata RPC è analoga a una chiamata di funzione. Quando viene eseguita una RPC, gli argomenti chiamanti vengono passati alla procedura remota e il chiamante aspetta che la procedura remota restituisca una risposta. La Figura 1 mostra il flusso di attività che avviene durante una chiamata RPC tra due sistemi collegati in rete. Il client esegue una chiamata di procedura che invia una richiesta al server e aspetta. Il thread del client viene bloccato finche non riceve una risposta. Quando la richiesta arriva, il server chiama una routine locale che esegue il servizio richiesto e invia la risposta al client. Dopo che la chiamata RPC è stata completata, il programma client continua la sua esecuzione.

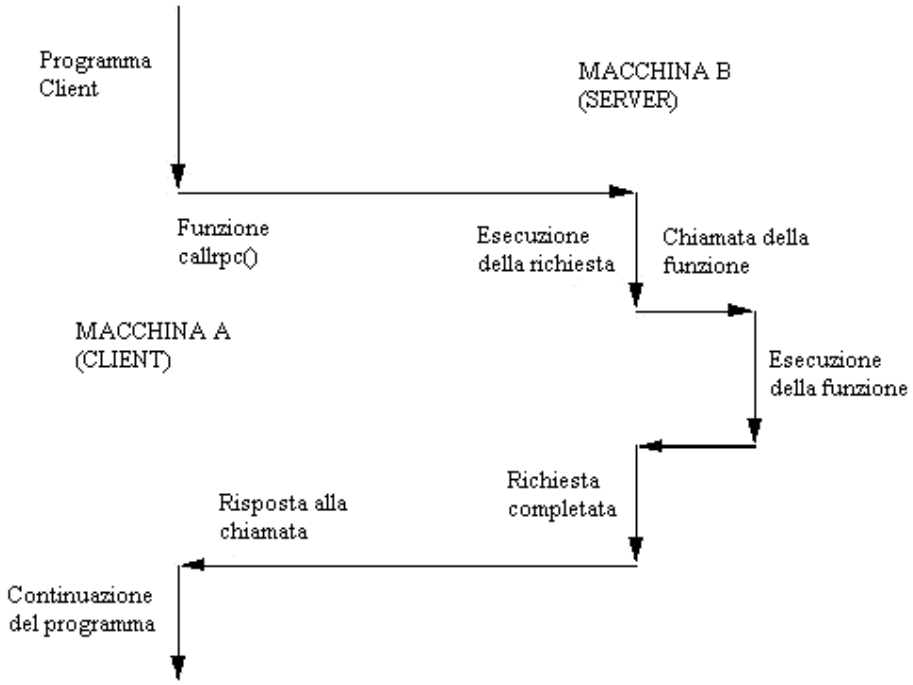

Figura 1 *Meccanismo di chiamata di procedura remota*.

La procedura remota viene individuata da tre elementi: il numero di programma, il numero di versione, il numero di procedura. Il numero di programma identifica un gruppo di procedure remote correlate, ognuna delle quali ha un numero di procedura univoco. Un programma può essere costituito da una o più versioni. Ogni versione consiste in una serie di procedure che possono essere chiamate a distanza. I numeri di versione consentono che più versioni di un'applicazione che usa RPC siano disponibili simultaneamente. Ciascuna versione contiene varie procedure che possono essere chiamate a distanza.

1.3 Sviluppo di applicazioni RPC

Prendiamo come esempio una ricerca client/server in un database posto su una macchina remota.
Potremmo usare telnet per attivare una shell remota ed eseguire il comando in questo modo, ciò però causerebbe vari problemi:

- l'esecuzione potrebbe essere molto lenta;
- è necessario un account login sulla macchina remota.

RPC invece prevede:

- la definizione di un server sulla macchina remota, in grado di rispondere a delle query;
- il recupero di informazioni eseguendo le query.

Per sviluppare l'applicazione RPC sono necessarie le seguenti operazioni:

- specificare il protocollo per la comunicazione client server;
- sviluppare il programma client ;
- sviluppare il programma server.

I programmi verranno compilati separatamente. Il protocollo di comunicazione viene implementato da codice generato automaticamente.

1.4 Definizione del protocollo

Il modo più semplice per definire il protocollo è usare un compilatore come `rpcgen`.
Per definire il protocollo si deve identificare il nome delle procedure di servizio e i tipi di dati dei parametri e gli argomenti di ritorno.
Il compilatore del protocollo legge la definizione e genera automaticamente il codice di comunicazione lato client (detto *stub*) e quello lato server (detto *skeleton*).
La definizione del protocollo viene scritta in un apposito linguaggio dichiarativo (linguaggio RPC o RPCL) che è molto simile alle direttive preprocessore.
`rpcgen` è un programma eseguibile autonomo che legge file speciali indicati da un prefisso .x.
Il seguente listato presenta un esempio di file *.x* per la definizione del protocollo di comunicazione.

```
const MAXNAMELEN = 255;                  /* lunghezza in byte della stringa */
typedef string nametype<MAXNAMELEN>;     /* voce directory */
typedef struct namenode *namelist;       /* link alla prossima struttura*/

/* Un nodo nella struttura della directory */
struct namenode {
   nametype name;             /* nome di un'entry della directory */
   namelist next;             /* prossima entry */
};

/*
* Valore di ritorno della funzione
* In questo esempio una union è usata per discriminare fra
* chiamate terminate con successo e chiamate con errore */
union readdir_res switch (int errno) {
   case 0:
     namelist list;                /* Nessun errore: restituisco la lista degli
                                   elementi della directory */
   default:
     void;                         /* In caso di errore, restituisce void */
   };
```

```
/* Definizione delle procedure del programma DIRPROG */

program DIRPROG {                          /* Nome del programma */
   version DIRVERS {                  /* Identif. della versione del programma*/
    readdir_res READDIR(nametype) = 1;    /* dichiarazione della proc. remota 1*/
    void DELDIR(nametype) = 2;            /* dichiarazione della proc. remota 2*/
   } = 1;                                 /* num. di versione */
} = 0x20000076;                           /* num. di programma */
```

Il protocollo specificato prevede la definizione di variabili, struct, union e procedure remote che saranno poi incluse e utilizzate dalle applicazioni client e server. In particolare, nella parte finale, definiamo la procedura remota `readdir`. E' importante notare come, in ambito RPCL, le variabili *char**, per non creare ambiguità, sono dichiarate semplicemente *string*. Per convenzione, i nomi delle procedure remote sono scritte in maiuscolo e convertite in minuscolo automaticamente durante la compilazione.
Per compilare un file RPCL basta usare semplicemente:

```
rpcgen rpcprog.x
```

Questo genererà quattro file:

`rpcprog_clnt.c`: lo stub client;
`rpcprog_svc.c`: lo stub server;
`rpcprog_xdr.c`: gli eventuali filtri XDR (*eXternal Data Representation*), di cui ci occuperemo in seguito;
`rpcprog.h`: il file d'intestazione necessario per qualsiasi filtro XDR.

La rappresentazione esterna dei dati (XDR) è un'astrazione dei dati necessaria per garantire che i parametri inviati dal client siano rappresentati correttamente sul server. Il client e il server possono infatti essere macchine di tipo diverso, con differenti CPU e sistemi operativi.

1.5 Definizione del codice delle applicazioni server e client

Cominciamo scrivendo il codice delle applicazioni client e server, che devono comunicare tramite procedure e tipi di dati specificati dal protocollo. Il lato server dovrà registrare le procedure che possono essere chiamate dal client, ricevere i parametri e ritornare i valori di ritorno opportuni.

1.6 Compilazione ed esecuzione dell'applicazione

Ora consideriamo il modello di compilazione completa necessario per eseguire un'applicazione RPC. I makefile sono utili per semplificare la compilazione delle applicazioni RPC, ma prima di creare un makefile bisogna capire il modello completo.
Supponiamo che il programma client si chiami `rpcprog`, che il programma server si chiami `rpcsvc.c`, che il protocollo sia stato definito in `rpcprog`, che lo strumento `rpcgen` sia stato usato per creare i file filtro, stub e skeleton chiamati rispettivamente: `rpcprog_clnt.c`, `rpcprog_svc.c`, `rpcprog_xdr.c`, `rpcprog.h`.
I programmi client e server devono includere `(#include "rpcprog.h");` dopo di che bisogna:

- compilare il codice client:

```
cc -c  rpcprog.c
```

- compilare lo stub client:

```
cc -c  rpcprog_clnt.c
```

- compilare il filtro XDR:

```
cc -c  rpcprog_xdr.c
```

- creare l'eseguibile client:

```
cc -o rpcprog rpcprog.o rpcprog_clnt.o rpcprog_xdr.c
```

- compilare le procedure server:

```
cc -c  rpcsvc.c
```

- compilare lo stub server:

```
cc -c  rpcprog_svc.c
```

- creare l'eseguibile server:

```
cc -o rpcsvc rpcsvc.o rpcprog_svc.o rpcprog_xdr.c
```

A questo punto basta eseguire i programmi `rpcprog` e `rpcsvc` rispettivamente sul client e sul server. Le procedure server devono essere registrate prima che il client possa chiamarle.
`rpcgen` automaticamente genera i file *_clnt*, *_svr*, *_xdr* e *.h*. Nei prossimi capitoli è spiegato passo per passo l'interfacce che implementano tali file e come un programmatore dovrebbe operare per implementarli manualmente.

1.7 Panoramica delle routine d'interfaccia

RPC ha più livelli d'interfaccia che corrispondono a un controllo più fine sulla modalità di chiamata. Questi livelli forniscono gradi diversi di controllo e corrispondono anche a diverse quantità di codice d'interfaccia da implementare. Vedremo ora alcune delle routine disponibili a ogni livello. Le interfacce standard possono essere di diversi livelli, infatti si dividono in interfacce di massimo livello (*top level*), di livello intermedio (*intermediate level*), di livello expert (*expert level*) e di livello più basso (*bottom level*). Queste interfacce danno allo

sviluppatore molto più controllo sui parametri di comunicazione, per esempio il trasporto da usare, il tempo di attesa prima della risposta agli errori, le richieste di ritrasmissione e così via. Per prime, consideriamo le interfacce semplificate che vengono usate per creare chiamate di procedure remote e specificano solo il tipo di trasporto da usare. Le routine di questo livello vengono usate per la maggior parte delle applicazioni.

1.7.1 Livello semplificato

`rpc_reg()`: registra una procedura come un programma RPC su tutti i trasporti del tipo specificato.

`rpc_call()`: chiama la procedura specificata sull'host remoto specificato.

`rpc_broadcast()`: invia un messaggio di chiamata attraverso tutti i trasporti di tipo specificato type.

1.7.2 Massimo livello

Al livello più alto, l'interfaccia è ancora semplice, ma il programma deve creare un client handle prima di fare una chiamata o creare un server handle prima di ricevere le chiamate. Se si desidera che l'applicazione funzioni su tutti i possibili trasporti, bisogna usare questa interfaccia. `clnt_create()`: creazione generica di client. Il programma comunica a `clnt_create()` dove è localizzato il server e il tipo di trasporto da usare.

`clnt_create_timed()`: simile a `clnt_create()`, ma consente al programmatore di specificare il tempo massimo consentito per ogni tipo di trasporto provato durante il tentativo di creazione.

`svc_create()`: crea i server handle per tutti i trasporti del tipo specificato. Il programma comunica a `svc_create()` quale funzione locale usare.

`clnt_call()`: il client chiama una procedura per inviare una richiesta al server.

1.7.3 Livello intermedio

L'interfaccia di livello intermedio di RPC consente di controllare i dettagli. I programmi scritti a questi livelli sono più complessi, ma eseguiti in modo più efficiente. Il livello intermedio consente di specificare il trasporto da usare.

`clnt_tp_create()`: crea un client handle per il trasporto specificato.

`clnt_tp_create_timed()`: simile al `clnt_tp_create()`, ma consente al programmatore di specificare il tempo massimo consentito.

`svc_tp_create()`: crea un server handle per il trasporto specificato.

`clnt_call()`: il client chiama una procedura per inviare una richiesta al server.

1.7.4 Livello expert

Il livello expert contiene un vasto repertorio di opzioni per specificare vari parametri di trasporto.

`clnt_tli_create()`: crea un client handle per il trasporto specificato.

`svc_tli_create()`: crea un server handle per il trasporto specificato.

`rpcb_set()`: chiama `rpcbind` per creare un mapping tra un servizio RPC e un indirizzo di rete.

`rpcb_unset()`: cancella un mapping impostato da `rpcb_set()`.

`rpcb_getaddr()`: chiama `rpcbind` per ottenere gli indirizzi di trasporto di servizi RPC specificati.

`svc_reg()`: associa il programma specificato e il numero di versione uniti alla routine specificata.

`svc_unreg()`: cancella un'associazione impostata da `svc_reg()`.

`clnt_call()`: il client chiama una procedura per inviare una richiesta al server.

1.8 Librerie d'interfaccia a RPC

Vedremo ora l'interfaccia C a RPC e come scrivere applicazioni di rete usando RPC. Per una spiegazione dettagliata delle routine nella libreria RPC, si consultino *rpc* e le pagine *man* correlate.

1.9 Interfaccia semplificata

L'interfaccia semplificata è il livello più facile da usare perché non richiede l'uso di nessun'altra routine RPC. Limita però il controllo dei meccanismi di comunicazione basilari. Lo sviluppo di programmi a questo livello è molto rapido ed è direttamente supportato dal compilatore `rpcgen`. Per la maggior parte delle applicazioni, questa interfaccia è sufficiente. Le routine di libreria dell'interfaccia semplificata forniscono un semplice accesso diretto alle facility RPC di controllo. Le routine come `rusers` sono presenti nella libreria di servizi RPC `librpcsvc`.
Il programma `rusers.c` qui di seguito chiama la routine della libreria RPC `rusers` per mostrare il numero di utenti su un host remoto.

Ecco il listato del programma `program.c`:

```
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>
#include <stdio.h>
```

```
/*
* un programma che chiama
* il servizio rusers()
*/

main(int argc,char **argv)

{
int num;
if (argc != 2) {
   fprintf(stderr, "usage: %s hostname\n",
   argv[0]);
   exit(1);
   }

if ((num = rnusers(argv[1])) < 0) {
   fprintf(stderr, "error: rusers\n");
   exit(1);
  }

fprintf(stderr, "%d users on %s\n", num, argv[1] );
exit(0);
}
```

La compilazione del programma viene fatta con:

```
cc program.c -lrpcsvc -lnsl
```

1.9.1 Il lato client

Sul lato client dell'interfaccia semplificata `rpc_call()` c'è solo una funzione, che ha nove parametri.

```
int rpc_call (char *host      /* Nome dell'host server */,
    u_long prognum            /* Numero del programma server */,
    u_long versnum            /* Numero di versione server */,
    xdrproc_t inproc          /* Filtro XDR per codificare argomenti */,
    char *in                  /* Puntatore agli argomenti */,
    xdr_proc_t outproc        /* Filtro per decodificare il risultato */,
    char *out                 /* Indirizzo per memorizzare il risultato*/,
    char *nettype             /* Per la selezione del trasporto */);
```

Questa funzione chiama la procedura specificata da `prognum`, `versum` e `procnum`. L'argomento che deve essere passato alla procedura remota è indicato dal parametro `in`, mentre `inproc` è il filtro XDR da usare per codificare questo argomento. Il parametro `out` è l'indirizzo in cui deve essere posto il risultato dalla procedura remota. `outproc` è un filtro XDR che decodificherà il risultato e lo porrà a questo indirizzo.
Il client si blocca su `rpc_call()` finché non riceve una risposta dal server. Se il server accetta, restituisce `RPC_SUCCESS` con il valore di zero. Restituirà un valore diverso da zero se la chiamata non ha avuto successo. Si può eseguire una forzatura di tipo di questo valore, portandolo al tipo `clnt_stat`, un tipo enumerativo definito nei file include RPC (<`rpc/rpc.h`>) e interpretato dalla funzione `clnt_sperrno()`. Questa funzione restituisce un puntatore a un messaggio d'errore standard RPC che corrisponde al codice d'errore. Nell'esempio, sono provati tutti i trasporti "visibili" elencati in `/etc/netconfig`. La modifica del numero di tentativi richiede l'uso dei livelli inferiori della libreria RPC. Risultati e argomenti multipli vengono gestiti raccogliendoli in strutture.
L'esempio per usare l'interfaccia semplificata ha il seguente aspetto:

```
#include <stdio.h>
#include <utmp.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>

/*
 * Un programma che chiama il programma RUSERSPROG RPC
 */

main(int argc, char **argv)

{
   unsigned long nusers;
   enum clnt_stat cs;
   if (argc != 2) {
     fprintf(stderr, "usage: rusers hostname\n");
     exit(1);
    }

   if( cs = rpc_call(argv[1], RUSERSPROG,
          RUSERSVERS, RUSERSPROC_NUM, xdr_void,
          (char *)0, xdr_u_long, (char *)&nusers,
          "visible") != RPC_SUCCESS ) {
              clnt_perrno(cs);
```

```
            exit(1);
          }

   fprintf(stderr, "%d users on %s\n", nusers, argv[1] );
   exit(0);
}
```

Poiché i tipi di dati possono essere rappresentati in modo diverso su macchine diverse, `rpc_call()` ha bisogno sia del tipo dell'argomento RPC sia di un puntatore all'argomento RPC stesso (come per il risultato). Per `RUSERSPROC_NUM`, il valore di ritorno è unsigned long, quindi il primo parametro di ritorno di `rpc_call()` è `xdr_u_long` (che è un unsigned long) e il secondo è `&nusers` (che punta a unsigned long storage). Poiché `RUSERSPROC_NUM` non ha argomenti, la funzione di codifica XDR di `rpc_call()` è `xdr_void()` e il suo argomento è `NULL`.

1.9.2 Il lato server

Il programma server che usa l'interfaccia semplificata è molto facile. Chiama semplicemente `rpc_reg()` per registrare la procedura e la routine di smistamento `svc_run()` per attendere l'arrivo di richieste.
`rpc_reg()` ha il seguente prototipo:

```
rpc_reg(u_long prognum        /* Numero del programma server */,
        u_long versnum        /* Numero della versione server */,
        u_long procnum        /* Numero della procedura server */,
        char *procname        /* Nome della funzione remota */,
        xdrproc_t inproc      /* Filtro per codificare gli argomenti */,
        xdrproc_t outproc     /* Filtro per decodif. il risultato */,
        char *nettype         /* Per la selezione del trasporto */);
```

`svc_run()` invoca procedure di servizio in risposta a messaggi di chiamate RPC. La routine `rpc_reg()`si occupa di decodificare gli argomenti della procedura remota e decodificare i risultati, usando i filtri XDR che sono stati specificati quando la procedura remota è stata registrata.
Ecco alcune osservazioni importanti sul programma server:

- La maggior parte delle applicazioni RPC segue la convenzione di denominazione di aggiungere un `_1` al nome della funzione. La `_n` sequenza viene aggiunta ai nomi delle procedure per indicare l'`n` del numero della versione del servizio.
- L'argomento e il risultato vengono passati come indirizzi. Questo vale per tutte le funzioni che sono chiamate a distanza. Se si passa `NULL` come risultato di una funzione,

nessuna risposta viene inviata al client. Si presuppone che non ci sia alcuna risposta da inviare.

- Il risultato deve esistere nello spazio di dati statici perché possa accedere al suo valore dopo che si è usciti dalla procedura. La funzione della libreria RPC che costruisce il messaggio di risposta RPC ha accesso al risultato e rimanda il valore al client.
- È consentito solo un singolo argomento; se ci sono più elementi dei dati dovrebbero essere incapsulati all'interno di una struttura che poi può essere passata come una singola entità.
- La procedura è registrata per ogni trasporto del tipo specificato. Se il parametro è di tipo (`char *)NULL`, la procedura è registrata per tutti i trasporti specificati in `NETPATH`.

Talvolta si può implementare un codice più veloce e più compatto di quello possibile con `rpcgen`. `rpcgen` gestisce i casi generici di generazione del codice. Il programma seguente è un esempio di routine di registrazione codificata manualmente. Registra una singola procedura e immette `svc_run()` nelle richieste di servizio.

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>

void *rusers();

main()
{
  if(rpc_reg(RUSERSPROG, RUSERSVERS,
        RUSERSPROC_NUM, rusers,
        xdr_void, xdr_u_long,
        "visible") == -1) {
           fprintf(stderr, "Couldn't Register\n");
            exit(1);
  }
  svc_run(); /* Bloccante, in questo caso non ritorna mai */
  fprintf(stderr, "Error: svc_run returned!\n");
  exit(1);
}
```

`rpc_reg()` può essere chiamato tutte le volte che serve per registrare programmi, versioni e procedure diversi.

1.10 Passaggio di tipi di dati arbitrari

I tipi di dati passati a e ricevuti da procedure remote possono essere una qualsiasi serie di tipi predefiniti oppure possono essere tipi definiti dal programmatore. RPC gestisce strutture di dati arbitrarie, indipendentemente dalle convenzioni di layout di struttura o dagli ordini dei byte su macchine diverse, convertendoli sempre in un formato di trasferimento standard chiamato XDR (*eXternal Data Representation, rappresentazione esterna dei dati*) prima di inviarli al livello di trasporto. La conversione da una rappresentazione macchina in XDR è chiamata serializzazione e il processo inverso deserializzazione. Gli argomenti del traduttore di `rpc_call()` e `rpc_reg()` possono specificare una procedura primitiva XDR, come `xdr_u_long()` o una routine fornita dal programmatore che elabora una struttura completa di argomenti. La routine di elaborazione degli argomenti deve accettare solo due argomenti: un puntatore al risultato e un puntatore all'handle XDR.

Sono disponibili le seguenti routine primitive XDR:

```
xdr_int() xdr_netobj() xdr_u_long() xdr_enum()
xdr_long() xdr_float() xdr_u_int() xdr_bool()
xdr_short() xdr_double() xdr_u_short() xdr_wrapstring()
xdr_char() xdr_quadruple() xdr_u_char() xdr_void()
```

La funzione non primitiva `xdr_string()`, che accetta più di due parametri, è chiamata da `xdr_wrapstring()`.
Per esempio di una routine, supponiamo che la struttura:

```
struct simple {
   int a;
   short b;
} simple;
```

contenga gli argomenti di una procedura. La routine XDR `xdr_simple()` traduce la struttura nel modo seguente:

```
#include <rpc/rpc.h>
#include "simple.h"

bool_t xdr_simple(XDR *xdrsp, struct simple *simplep)

{
   if (!xdr_int(xdrsp, &simplep->a))
      return (FALSE);
```

```
   if (!xdr_short(xdrsp, &simplep->b))
      return (FALSE);
   return (TRUE);
}
```

Una routine equivalente può essere creata automaticamente da `rpcgen`. La routine XDR restituisce un valore diverso da zero (un TRUE) se termina con successo; diversamente restituisce zero.
Per strutture di dati più complesse si usano le seguenti routine pronte XDR:

```
xdr_array() xdr_bytes() xdr_reference()
xdr_vector() xdr_union() xdr_pointer()
xdr_string() xdr_opaque()
```

Per esempio, per inviare un array di dimensioni variabili di interi, impaccato in una struttura contenente l'array e la sua lunghezza, si usa la seguente dichiarazione:

```
struct varintarr {
int *data;
int arrlnth;
} arr;
```

Si traduce l'array con `xdr_array()`, come mostrato qui di seguito:

```
bool_t xdr_varintarr(XDR *xdrsp, struct varintarr *arrp)
{
    return(xdr_array(xdrsp, (caddr_t)&arrp->data,
              (u_int *)&arrp->arrlnth, MAXLEN, sizeof(int), xdr_int));
}
```

Gli argomenti di `xdr_array()` sono l'handle XDR, un puntatore all'array, un puntatore alle dimensioni dell'array, le dimensioni massime dell'array, le dimensioni di ogni elemento dell'array e un puntatore alla routine XDR per tradurre ogni elemento dell'array. Se le dimensioni dell'array sono note anticipatamente si usa `xdr_vector()` perché più efficiente:

```
int intarr[SIZE];
bool_t xdr_intarr(XDR *xdrsp, int intarr[])
{
      return (xdr_vector(xdrsp, intarr, SIZE, sizeof(int), xdr_int));
}
```

XDR converte le quantità in multipli di 4 byte durante la serializzazione. Per array di caratteri, ogni carattere occupa 32 bit. `xdr_bytes()` impacca i caratteri e ha quattro parametri simili ai primi quattro parametri di `xdr_array()`.
Le stringhe null-terminated (come quelle del linguaggio) vengono tradotte da `xdr_string()`; è come `xdr_bytes()` con nessun parametro length. Alla serializzazione prende la lunghezza della stringa da `strlen()` e alla deserializzazione crea una stringa null-terminated.
`xdr_reference()` chiama le funzioni incorporate `xdr_string()` e `xdr_reference()`, che traduce i puntatori per passare una stringa e la struttura degli esempi precedenti. Un utilizzo d'esempio di `xdr_reference()` è il seguente:

```
struct finalexample {
    char *string;
    struct simple *simplep;
} finalexample;

bool_t xdr_finalexample(XDR *xdrsp, struct finalexample *finalp)

{  if (!xdr_string(xdrsp, &finalp->string, MAXSTRLEN))
        return (FALSE);
   if (!xdr_reference( xdrsp, &finalp->simplep, sizeof(struct
        simple), xdr_simple))
       return (FALSE);
    return (TRUE);
}
```

Si tenga presente che qui invece di `xdr_reference()` si sarebbe potuto chiamare `xdr_simple()`.

1.11 Sviluppo di applicazioni RPC di alto livello

Ora vedremo alcune altre funzioni e parleremo di come sviluppare un'applicazione usando le routine RPC di alto livello. Per questo esamineremo un esempio.
Svilupperemo un'utility per leggere una directory remota; per comprendere il problema affronteremo prima quello di leggere una directory locale.
Consideriamo il programma costituito da due file:

- `lls.c`: il programma principale che chiama una routine locale `read_dir.c`

```
/*
 * ls.c: esegue la lista della directory – senza RPC
 */
#include <stdio.h>
```

```
#include <strings.h>
#include "rls.h"

main (int argc, char **argv)

{
        char    dir[DIR_SIZE];

        /* chiamata della procedura locale */
        strcpy(dir, argv[1]); /* char dir[DIR_SIZE] viene e va...*/
        read_dir(dir);

        /* stampa dei risultati! */
        printf("%s\n", dir);

        exit(0);
}
```

- `read_dir.c`: il file che contiene la routine **locale** `read_dir()`.

```
/* nota – le chiamate di procedure locali compatibili con RPC prendono un input
e ritornano un output designati tramite puntatori. I valori di ritorno devono
puntare a dati statici per garantire che restino accessibili dopo la fine della
procedura locale. */

#include <stdio.h>
#include <sys/types.h>
#include <sys/dir.h>
#include "rls.h"

read_dir(char *dir)
   /* char dir[DIR_SIZE] */
{
        DIR * dirp;
        struct direct *d;
         printf("beginning ");

        /* apertura della directory */
        dirp = opendir(dir);
        if (dirp == NULL)
                return(NULL);
```

```
        /* inserimento dei nomi di file nel buffer dir */
        dir[0] = NULL;
        while (d = readdir(dirp))
                sprintf(dir, "%s%s\n", dir, d->d_name);

        /* ritorno del risultato */
        printf("returning ");
        closedir(dirp);
        return((int)dir);
}
```

Il file d'intestazione `rls.h` contiene solo la definizione seguente (almeno per ora)

```
#define DIR_SIZE 8192
```

Questo programma locale verrebbe compilato come segue:

```
cc lls.c read_dir.c -o lls
```

A questo punto vediamo come modificare questo programma per lavorare su una rete in modo che ci permetta di esaminare una directory di un server remoto attraverso una rete.
Saranno necessarie le seguenti fasi:

- Dovremo convertire il `read_dir.c` per l'esecuzione sul server.
- Dovremo registrare il server e la routine `read_dir()` sul server.
- Il client `lls.c` dovrà chiamare la routine come una procedura remota.
- Dovremo definire il protocollo per la comunicazione tra i programmi client e server.

1.12 Definizione del protocollo

Possiamo usare semplici stringhe null-terminated per passare e ricevere il nome della directory e il suo contenuto. Inoltre, possiamo incorporare il passaggio di questi parametri direttamente nel codice del server e del client.
Dobbiamo quindi specificare i numeri di versione, procedura e programma per client and server. Questo può essere eseguito automaticamente usando `rpcgen` o basandosi sulle macro predefinite nell'interfaccia semplificata. Qui li specificheremo manualmente.
Il server e il client deve essere d'accordo **fin dall'inizio** su quali indirizzi logici useranno. I numeri di programma vengono definiti in un modo standard:

- 0x00000000 - 0x1*FFFFFFF*: Definito da Sun
- 0x20000000 - 0x3*FFFFFFF*: Definito dall'utente
- 0x40000000 - 0x5*FFFFFFF*: Transitorio
- 0x60000000 - 0x*FFFFFFFF*: Riservato

Sceglieremo semplicemente un **valore definito dall'utente** per il nostro numero di programma. I numeri di versione e procedura vanno invece impostati secondo una regola standard.
Abbiamo ancora da eseguire la definizione `DIR_SIZE` che è indispensabile poiché le dimensioni del buffer della directory sono richieste sia dai programmi client sia dai programmi server.
Il nostro nuovo file `rls.h` contiene:

```
#define DIR_SIZE 8192
#define DIRPROG ((u_long) 0x20000001) /* numero dei programmi server */
#define DIRVERS ((u_long) 1)    /* numero della versione del programma */
#define READDIR ((u_long) 1)    /* numero della procedura per la ricerca */
```

1.13 Condivisione dei dati

All'inizio della dispensa abbiamo detto che possiamo passare i dati dal client al server come semplici stringhe. Dobbiamo però definire una routine di filtro XDR `xdr_dir()` per convertire i dati nel formato condiviso XDR. Si tenga presente che può essere gestito solo un codice di codifica e decodifica. Questo è facile usando la routine di conversione standard `xdr_string()`.
Il file XDR, `rls_xrd.c`, è come segue:

```
#include <rpc/rpc.h>

#include "rls.h"

bool_t xdr_dir(XDR *xdrs, char *objp)

      { return ( xdr_string(xdrs, &objp, DIR_SIZE) ); }
```

1.13.1 Il lato server

Possiamo usare il file `read_dir.c` originale. Tutto ciò che dobbiamo fare è registrare la procedura e attivare il server.
La procedura si registra con la funzione `registerrpc()`. Ecco il prototipo:

```
registerrpc(u_long prognum          /* Numero del programma server */,
        u_long versnum              /* Numero di versione server */,
```

```
        u_long procnum              /* Numero di procedura server */,
        char *procname              /* Nome della funzione remota */,
        xdrproc_t inproc            /* Filtro per codificare gli argomenti*/,
        xdrproc_t outproc           /* Filtro per decodificare il risultato*/ );
```

I parametri sono definiti in modo simile a come lo sono nella funzione d'interfaccia semplificata `rpc_reg`. Abbiamo già discusso l'impostazione del parametro con i file d'intestazione `rls.h` del protocollo e il file del filtro XDR `rls_xrd.c`. Anche la routine `svc_run()` è già stata esaminata precedentemente.
Il codice completo `rls_svc.c` è come segue:

```
#include <rpc/rpc.h>
#include "rls.h"

main()
{
        extern bool_t xdr_dir();
        extern char * read_dir();

        registerrpc(DIRPROG, DIRVERS, READDIR,
                    read_dir, xdr_dir, xdr_dir);
        svc_run();
}
```

1.13.2 Il lato client

Dal lato client dobbiamo semplicemente chiamare la procedura remota. La funzione `callrpc()`, per farlo, ha il seguente prototipo:

```
callrpc(char *host            /* Nome dell'host server */,
    u_long prognum            /* Numero del programma server */,
    u_long versnum            /* Numero della versione server */,
    char *in                  /* Puntatore all'argomento */,
    xdrproc_t inproc          /* Filtro XDR per codificare l'argomento */,
    char *out                 /* Indirizzo per memorizzare il risultato */
    xdr_proc_t outproc        /* Filtro per decodificare il risultato */
);
```

Chiamiamo una funzione locale `read_dir()` che usa `callrpc()` per chiamare la procedura remota che è stata registrata come `READDIR` al server.

Il programma `rls.c` completo è come segue:

```
/*
 * rls.c: client del listato della directory remota
 */
#include <stdio.h>
#include <strings.h>
#include <rpc/rpc.h>
#include "rls.h"

main (argc, argv)
int argc; char *argv[];
{
         char    dir[DIR_SIZE];

        /* chiamata alla procedura remota se registrata */
        strcpy(dir, argv[2]);
        read_dir(argv[1], dir); /* read_dir(host, directory) */

        /* stampa dei risultati! */
        printf("%s\n", dir);

        exit(0);
}

read_dir(host, dir)
char   *dir, *host;
{
        extern bool_t xdr_dir();
        enum clnt_stat clnt_stat;

        clnt_stat = callrpc (host, DIRPROG, DIRVERS, READDIR,
                             xdr_dir, dir, xdr_dir, dir);
        if (clnt_stat != 0) clnt_perrno (clnt_stat);
}
```