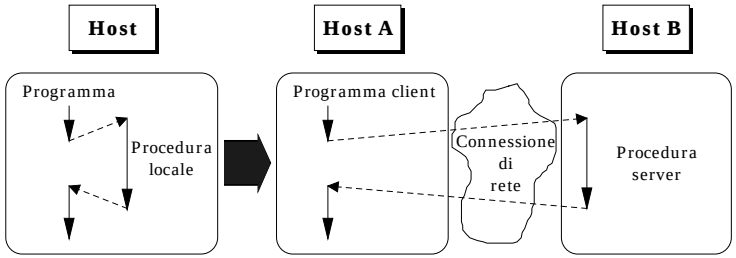
REMOTE Procedure Call

Possibilità di invocare una procedura non locale

operazione che interessa un nodo remoto e ne richiede un servizio

MODELLO CLIENTE/SERVITORE
invocazione di un servizio remoto
con **parametri tipati**, valore di **ritorno**

Modello di base: P.B. Hansen (Distributed Processing)
Nelson Birrel



PROPRIETÀ

- *eterogeneità*:
  dati diversi su macchine diverse
- *semantica diversa*
  del modello locale esteso in modo remoto
- *efficienza*
  implementazioni efficienti del modello
  reti locali ad elevata velocità di trasferimento dati
- *autenticazione*
  controllo di accesso da parte di utenti autorizzati

Requisiti per
IMPLEMENTAZIONE RPC

Il supporto scambia **messaggi** per consentire

- *identificazione* dei messaggi di chiamata e risposta
- *identificazione* unica della procedura remota
- *autenticazione* del chiamante e del servitore

oltre a
amministrazione della rete e
gestione di alcuni tipi di **errori** dovuti alla distribuzione

- *implementazione errata*
- *errori dell'utente*
- *roll-over* (ritorno indietro)

Il meccanismo RPC deve gestire i seguenti eventi anomali:

- *incongruenze di protocollo RPC*
- *incongruenze fra versioni di programmi*
- *errori di protocollo* (ad esempio parametri errati)
- *autenticazione fallita sulla procedura remota* e identificazione del motivo del rifiuto dell'accesso
- *altre ragioni per cui la procedura remota non viene chiamata o eseguita*

SEMANTICA di INTERAZIONE

A fronte di **possibilità di guasto**, il cliente può controllare o meno il servizio

maybe
at least once (***SUN RPC***)
at most once
exactly once

Per il **parallelismo** e la **sincronizzazione** possibile

operazioni per il servitore
sequenziali (***SUN RPC***)
paralleli

operazioni per il cliente
sincrone (***SUN RPC***)
asincrone

IMPLEMENTAZIONE

RPC meccanismo classificabile in due categorie secondo il grado di **trasparenza**

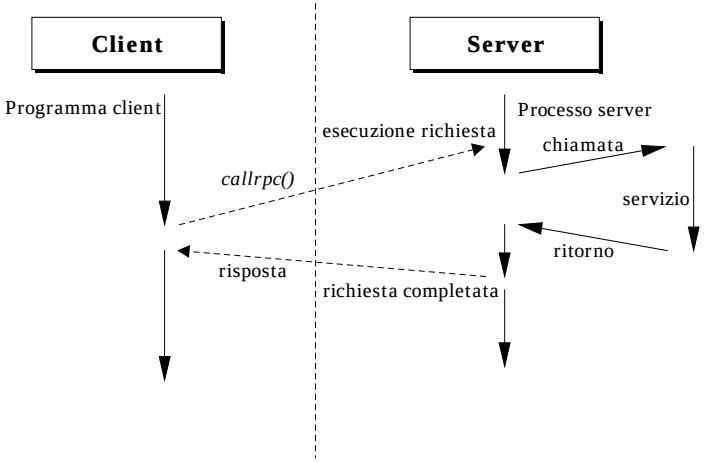
- **O**pen **N**etwork **C**omputing (Sun Microsystems)
- **N**etwork **C**omputing **A**rchitecture (Apollo)

Open Network Computing (ONC)

chiamata RPC Sun ***(diversa dalla locale)***
primitiva *callrpc()*

parametri necessari
nome del nodo remoto
identificatore della procedura da eseguire
specifiche di **trasformazione** degli argomenti (*eXternal Data Representation* **XDR**)

Schema del modello ONC



NON TRASPARENZA

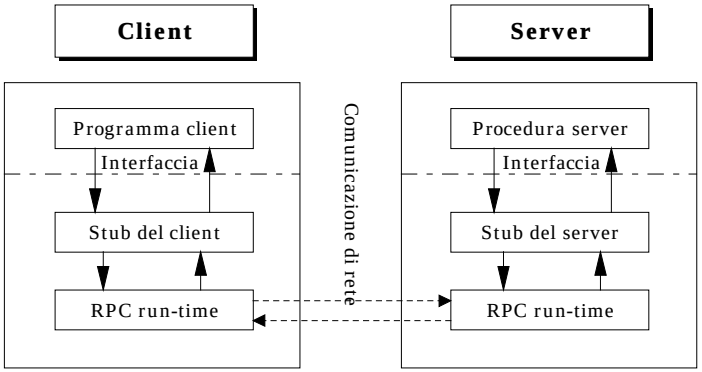
HP-UX, UNIX compatibili, Novell Netware

Network Computing Architecture (NCA)

Ai due lati della comunicazione, **client** e **server**
routines *stub* per ottenere la **trasparenza**

Chiamate locali allo stub

Schema del modello NCA



Gli stub sono forniti dall'implementazione
generati automaticamente
Le parti di programma sono "del tutto" ***inalterate***
ci si dirige verso lo stub che nasconde le operazioni

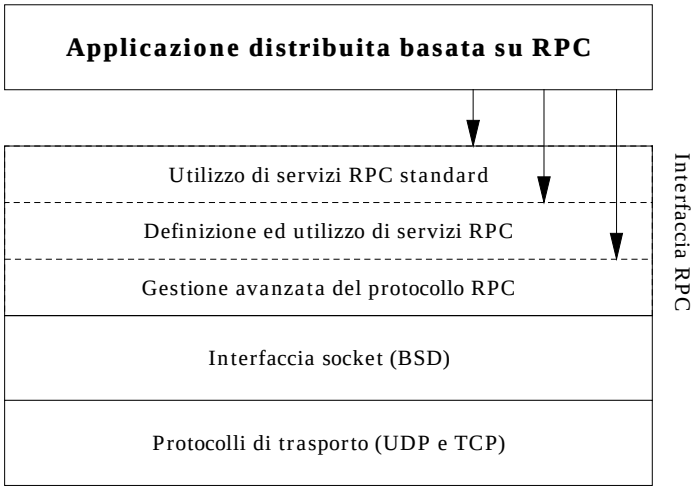
COSTO: due messaggi per RPC
Mercury ottimizza i messaggi con **stream di chiamate**

UNIX

le RPC sfruttano i servizi delle socket

sia TCP per stream
servizi con connessione

sia UDP per datagrammi (**default SUN**)
servizi senza connessione



SUN RPC

- Un programma contiene più procedure remote che possono essere invocate
  Sono previste versioni multiple delle procedure

- Un unico argomento in ingresso ed in uscita per ogni invocazione

Semantica e controllo concorrenza

Mutua esclusione garantita nell'ambito di un programma:
una sola invocazione per volta

Semantica e affidabilità

Uso di protocollo UDP
Semantica at-least-once

Vengono fatte un numero a default di ritrasmissioni dopo un *intervallo di time-out* (in genere 4 secondi)

Non si prevede alcuna *concorrenza* a default nell'ambito dello stesso programma server

Possibilità di **deadlock** ==> se un server in RPC richiede, a sua volta, un servizio al programma chiamante

Identificazione delle procedure remote

Messaggio RPC deve contenere
numero di programma
numero di versione
numero di procedura
per la identificazione globale

Numeri di programma (32 bit)

- **0 - 1FFFFFFFh** predefinito Sun
  Applicazioned'interesse comune
- **20000000h - 3FFFFFFFh** definibile dall'utente
  applicazioni debug dei nuovi servizi
- **40000000h - 5FFFFFFFh** riservato alle applicazioni
  per generare dinamicamente numeri di programma
- Altri gruppi riservati per estensioni

notare: **32** bit per il numero di programma
numero delle porte **16** bit
soluzione **Aggancio DINAMICO**

Autenticazione

sicurezza ==>
identificazione del client presso il server e viceversa sia in chiamata sia in risposta

IMPLEMENTAZIONE di RPC

Messaggi

```
enum msg_type {CALL = 0,  REPLY = 1};
struct rpc_msg {
    unsigned int xid; /* numero unico di identificazione */
    union switch (msg_type mtype) {
        case CALL:      struct call_body cbody;
        case REPLY:     union reply_body rbody;
    } body;
};
```

Un messaggio di **chiamata** contiene:

- *identificazione dei messaggi di chiamata e risposta*
  campo xid
- *corpo del messaggio di chiamata* strutturato

In un programma/processo anche più procedure

```
struct call_body {
/* versione protocollo RPC*/
        unsigned int rpcvers;
/* identificazione unica della procedura remota*/
        unsigned int prog;  /* Programma */
        unsigned int vers;  /* Versione */
        unsigned int proc;  /* Procedura */
/* autenticazione del chiamante e del servitore della
   chiamata*/
        opaque_auth cred;
        opaque_auth verf;
/* da qui iniziano gli argomenti effettivi della chiamata RPC */
            };
```


In un messaggio di **risposta**:

- *identificazione dei messaggi di chiamata e risposta*:
  campo xid
- *corpo del messaggio di risposta*:

```
enum reply_stat {MSG_ACCEPTED = 0,MSG_DENIED = 1}

union reply_body switch (reply_stat stat)
    {   case MSG_ACCEPTED:
            struct accepted_reply areply;
/* risposta e suo tipo. Nel caso di programma o procedura non
   esistente e di argomenti errati, la risposta è nulla. */
        case MSG_DENIED:
            struct rejected_reply rreply;
/* indicazione di eventi non gestibili */
    };
```

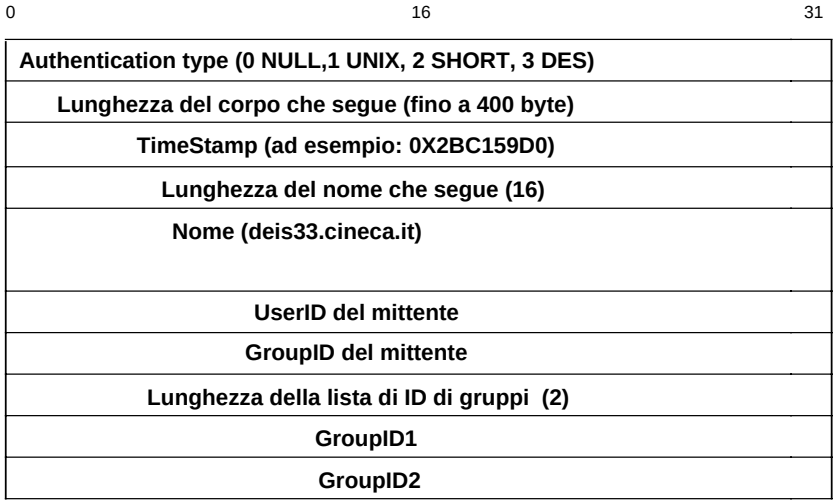
Fino al ritorno al programma cliente, il **processo cliente** è **bloccato** in attesa della risposta

Naturalmente, più chiamate RPC possono manifestarsi in parallelo su un nodo servitore

SERVIZI PARALLELI??

Formato di Chiamata RPC

Formato di autenticazione



Controllo degli accessi (autenticazione)

- ***Null authentication***
- ***UNIX authentication***: modalità UNIX

```
struct auth_unix {
     unsigned int stamp;
     char machine[255];
     unsigned int uid;
     unsigned int gid;
     unsigned int gids[16];
              };
```

La struttura del controllore è **Null authentication**

Problemi

- identificazione troppo legata a UNIX
- non universalità dell'identificazione legata a UID e GID
- mancanza di un controllore passato al partner

- ***Data Encryption Standard authentication***

  client come stringa di caratteri *netname*
  identificazione unica del cliente

controllore tempo del sistema client cifrato

Meccanismo

. il client esegue la cifratura con una *conversation key* legata alla sessione RPC
  La chiave viene generata dal client e passata al server all'atto della prima chiamata remota
. il client ed il server devono avere la stesso tempo
  Uso di *Network Time Protocol* o richiesto al server con la prima chiamata remota

Primo livello di RPC

Uso di funzioni dirette pronte per essere invocate da un nodo ad un altro

Routine di libreria RPC	Descrizione
rnusers()	Fornisce il numero di utenti di un nodo
rusers()	Fornisce informazioni sugli utenti di un nodo
havedisk()	Determina se un nodo ha un'unità disco
rstat()	Ottiene dati sulle prestazioni verso un nodo
rwall()	Invia al nodo un messaggio
getmaster()	Ottiene il nome del nodo master per NIS
getrpcport()	Ottiene informazioni riguardo agli indirizzi TCP legati ai servizi RPC
yppasswd()	Aggiorna la parola d'accesso in NIS

Numeri di programma noti

```
portmap 100000  (port mapper)
rstat   100002  (demone rstad)
rusers  100002  (demone rusersd)
nfs     100003  (demone nfsd)
...
```


utilizzo di servizi RPC standard

```
/* utilizzare la libreria corretta:
   ad esempio rpcsvc in fase di linking */
#include <stdio.h>
int rnusers(); /* dichiarazione di rnusers  */

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

if (argc<2)
{fprintf(stderr,"uso:%s hostname\n", argv[0]);
 exit(1);      }
if ((num=rnusers(argv[1]))<0)
    {
      fprintf(stderr,"errore: rnusers\n");
      exit(1);
    }
printf("%d utenti su %s\n",num,argv[1]);
exit(0);
}
```

Si possono così usare direttamente le funzioni remote se sono riconosciute

Viene invocato direttamente il server richiesto
In realtà, si veda dopo cosa c'è sotto...

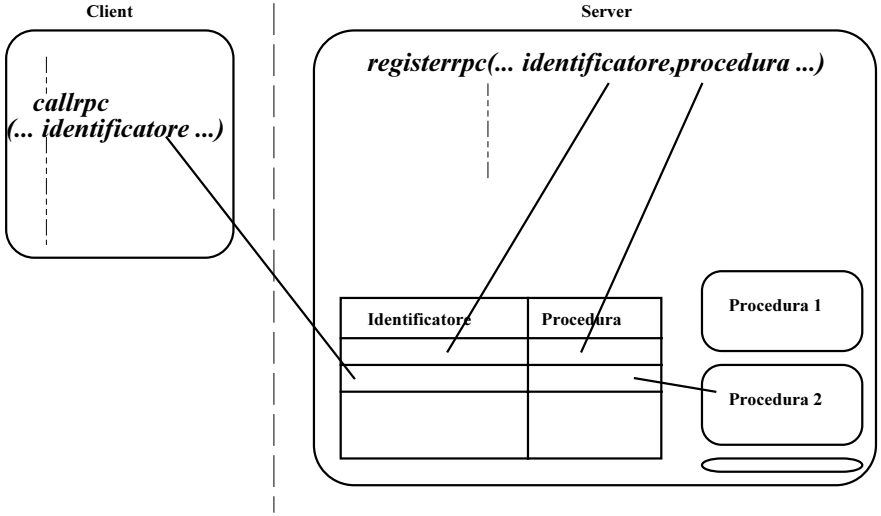
Livello intermedio RPC

Definizione ed utilizzo di servizi (ONC)

Due primitive: *callrpc()* e *registerrpc()*

CLIENT - callrpc() chiamata esplicita al meccanismo RPC e provoca l'esecuzione della procedura remota

SERVER - registerrpc() associa un identificatore unico alla procedura remota implementata nell'applicazione



Azioni realmente compiute per ogni invocazione RPC dalla parte del servitore

ricerca in tabella

messaggio al programma ritrovato

Richiesta di esecuzione della RPC (CLIENT)

primitiva callrpc
due parametri uno argomento e altro risultato,

```
int callrpc (remotehost, n_prog, n_vers,
   n_proc, xdr_arg, arg, xdr_res, res)
char       *remotehost;
u_long     n_prog,n_vers,n_proc;
xdrproc_t  xdr_arg;
char       *arg;
xdrproc_t  xdr_res;
char       *res;
```

remotehost == nome del nodo remoto
{**n_prog n_vers n_proc**}
== identificazione della procedura
xdr_arg == tipo di argomento
arg == argomento (passaggio per valore)
xdr_ares == tipo di risultato (per la funzione XDR)
res == risultato

Problemi nel passaggio di strutture a lista
Il ricevente deve ricostruire la struttura dinamica
(con funzioni opportune qui specificate in **xdr**)

La callrpc restituisce **successo** (=0) o
la causa di **insuccesso**
Ritorno intero appartenente all'insieme di valori della struttura *enum clnt_stat* definita nel file <rpc/clnt.h>

ERRORI o SUCCESSO

```
enum clnt_stat {RPC_SUCCESS=0, /* call succeeded */
/* local errors      */
RPC_CANTENCODEARGS=1,  /* can't encode args */
RPC_CANTDECODERES=2,     /* can't decode results */
RPC_CANTSEND=3,              /* failure in sending call */
RPC_CANTRECV=4,              /* failure in receiving result */
RPC_TIMEDOUT=5,                   /* call timed out */
/* remote errors */
RPC_VERSMISMATCH=6, /* rpc versions not compatible */
RPC_AUTHERROR=7,      /* authentication error */
RPC_PROGUNAVAIL=8,   /* program not available */
RPC_PROGVERSMISMATCH=9,
                /* program version mismatched */
RPC_PROCUNAVAIL=10,  /* procedure unavailable */
RPC_CANTDECODEARGS=11,/* decode args error */
RPC_SYSTEMERROR=12, /* generic "other problem" */
/* callrpc errors */
RPC_UNKNOWNHOST=13,      /* unknown host name */
RPC_UNKNOWNPROTO=17,    /* unkown protocol */
/* create errors */
RPC_PMAPFAILURE=14,  /* pmapper failed in its call */
RPC_PROGNOTREGISTERED=15,
                    /* remote program not registered */
/* unspecified error */
RPC_FAILED=16
#ifdef KERNEL
, RPC_INTR=18  /* Used for interrupts in kernel calls*/
#endif /*  KERNEL */
};
```


Possibilità di terminazione per time-out

Prima di dichiarare time-out, l'interfaccia avanzata esegue alcune ritrasmissioni in modo trasparente

Esempi di chiamate remote

vedi funzione **rnusers**()
conoscendo l'identificatore del servizio in <rpcsvc/rusers.h>

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

main(argc,argv)
int argc;  int *argv[];
{  unsigned long nusers;

if (argc<2)
{fprintf(stderr,"uso: %s hostname\n",argv[0]); exit(1); }
if ( callrpc ( argv[1], RUSERSPROG, RUSERSVERS,
  RUSERSPROC_NUM, xdr_void, 0, xdr_u_long, &nusers)
     != 0)
      {fprintf(stderr,"error: callrpc\n");
        exit(1);    }
printf("%d users on %s\n",  nusers,   argv[1]);
exit(0);
}
```

Semplice esempio di applicazione
sia client
sia server che implementa il servizio

livello intermedio d'accesso a RPC:

realizzazione di servizi remoti

Identificatore del servizio d'esempio scelto fra quelli disponibili per l'utente e noto ad entrambi i programmi

```
#include <stdio.h>
#include <rpc/rpc.h>
#define RPC5 (unsigned long) 0x20000015
#define VER5 (unsigned long) 1
#define PRC5 (unsigned long) 1
main(argc,argv)
int argc;  char *argv[];
{    unsigned long b;     unsigned long a=3;
if (argc<2)
     {fprintf(stderr,"uso:%s hostname\n",argv[0]); exit(1);}
if (callrpc (argv[1], RPC5, VER5, PRC5, xdr_u_long, &a,
                              xdr_u_long, &b)!=0)
          {fprintf(stderr,"errore:callrpc\n");   exit(1);}
printf("Risultato: %d\n", b);  exit(0);
}
```

Operazione aritmetica sull'argomento
intero in ingresso e
risultato intero in uscita
Implementazione del servizio remoto riportata dopo

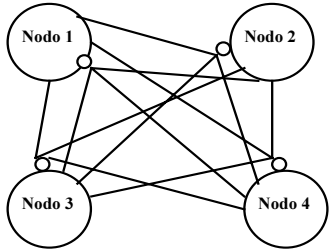
Omogeneità dei dati

Per comunicare tra nodi eterogenei due soluzioni:

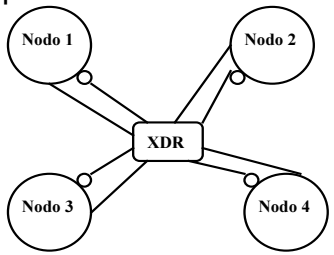
1. dotare ogni nodo di **tutte le funzioni di conversione** possibili per ogni rappresentazione dei dati
2. concordare un **formato comune** di rappresentazione dei dati: ogni nodo possiede le funzioni di conversione da/per questo formato

La prima ==> **elevata performance**
La seconda ==> **implementazione di un minore numero di funzioni di conversione**

N nodi eterogenei
nel primo caso le funzioni di conversione sono N*(N-1)
nel secondo sono N verso ed N per il formato comune



Sono necessarie 12 funzioni di conversione del formato di dati

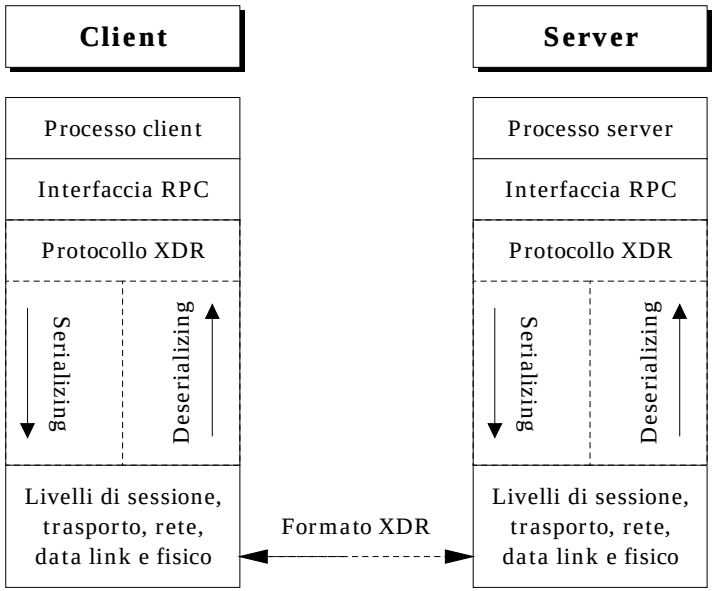


Sono necessarie 8 funzioni di conversione del formato di dati

Standard (modello OSI) porta alla scelta del secondo metodo

- **XDR** (Sun Microsystems)
- **Abstract Syntax Notation 1** (**ASN.1**) definito in ambito OSI con la sigla X.409.

eXternal Data Representation (XRD)



Funzioni svolte dal protocollo XDR

collocazione in OSI

XDR procedure *built-in* di conversione, relative a
- tipi atomici predefiniti e
- tipi standard (costruttori riconosciuti)

la primitiva ***callrpc***
accetta solo un argomento ed un solo risultato ==>
necessità di definire una struttura che le raggruppa

Funzioni built-in di conversione

Funzione built-in	Tipo di dato convertito
xdr_bool()	Logico
xdr_char() *xdr_u_char()*	Carattere
xdr_short() *xdr_u_short()*	Intero a 16 bit
xdr_enum()	Enumerazione
xdr_float()	Virgola mobile
xdr_int() *xdr_u_int*	Intero
xdr_long() *xdr_u_long()*	Intero a 32 bit
xdr_void()	Nullo
xdr_opaque()	Opaco (byte senza significato particolare)
xdr_double()	Doppia precisione

Funzioni per tipi composti

Funzione	Tipo di dato convertito
xdr_array()	Vettori con elementi di tipo qualsiasi
xdr_vector()	Vettori a lunghezza fissa
xdr_string()	Sequenza di caratteri con NULL
xdr_bytes()	Vettore di bytes senza terminatore
xdr_reference()	Riferimento ad un dato
xdr_pointer()	Riferimento ad un dato, incluso NULL
xdr_union()	Unioni

Ad esempio, booleani

file <rpc/xdr.h>
bool_t **xdr_int** (xdrs,ip)
    XDR *xdrs; int *ip;

Le funzioni xdr ritornano valore vero se la conversione ha successo

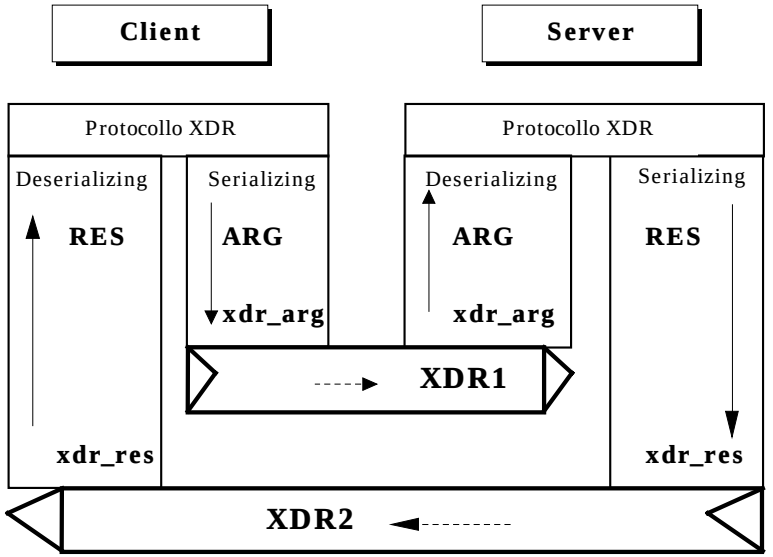
Anche con tipi definiti dall'utente

```
/* creazione di una struttura per il passaggio a callrpc() */
struct simple {
                int a;   short b;
                } simple;


/* procedura XDR mediante descrizione con funzioni built-in */
#include <rpc/rpc.h>    /* include xdr.h */
bool_t  xdr_simple (xdrsp, simplep)
  /* la struttura *xdrsp rappresenta il tipo in formato XDR */
XDR *xdrsp;
  /* la struttura *simplep rappresenta il formato interno */
struct simple *simplep;

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


Stream XDR

Per ogni informazione da trasmettere
    due trasformazioni

Sulla rete il solo formato **XDR**



Flussi XDR

Ogni nodo deve provvedere solamente le proprie funzionalità di trasformazione
    dal formato **locale** a quello **standard**
    dal formato **standard** a quello **locale**

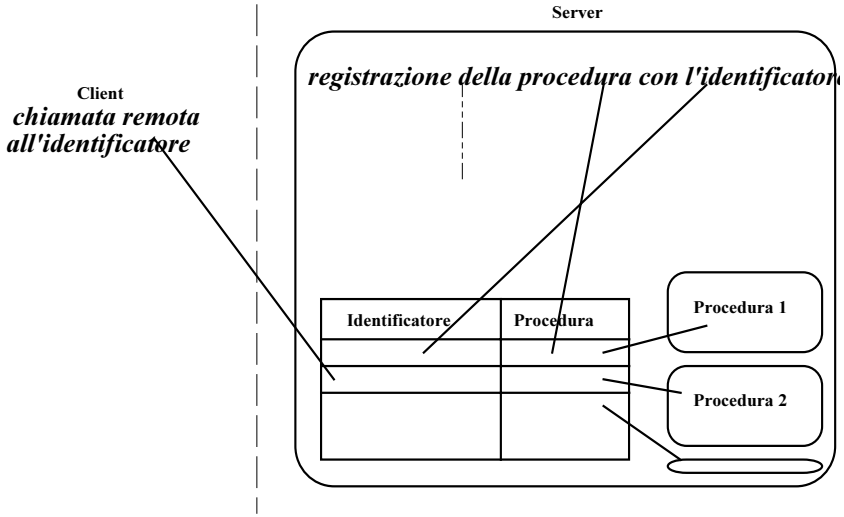
Le funzioni XDR possono essere usate anche per **trasformazione dei dati** (e non associate alla comunicazione)

Registrazione della procedura remota

SERVER

Un servizio registrato può essere invocato

al servizio viene associato un identificatore strutturato secondo il protocollo RPC



Il server deve registrare la **procedura** nella **tabella dinamica** dei **servizi** del nodo

Una entry per procedura

primitiva di registrazione

int **registerrpc** (n_prog, n_vers, n_proc, proc_name, xdr_arg, xdr_res)

/ Identificatore per il protocollo RPC ==>*
*numero di programma, versione e procedura */*

u_long n_prog, n_vers, n_proc;

/ puntatore alla procedura locale che implementa il servizio remoto */*

char *(* proc_name) ();

/ tipo di argomento della procedura e tipo del risultato */*

xdrproc_t xdr_arg, xdr_res;

{**n_prog n_vers n_proc**}
== identificazione della procedura

proc_name == puntatore alla procedura da attivare

xdr_arg == tipo di argomento

xdr_ares == tipo di risultato (per la funzione XDR)

La **registerrpc** viene eseguita dal server che la vuole rendere nota ed invocabile da clienti remoti

Una entry per ogni registrazione

int **svc_run()**

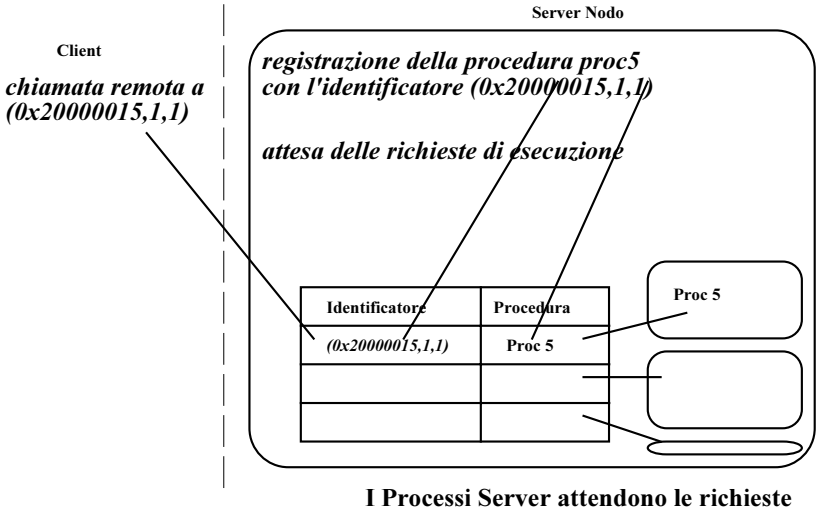
Il processo del programma corrente si mette in attesa indefinita di una richiesta per fornire i servizi

Esempio di registrazione di un servizio

```
#include <stdio.h>
#include <rpc/rpc.h>
#define RPC5 (u_long) 0x20000015
#define VER5 (u_long) 1
#define PRC5 (u_long) 1

/* dichiarazione della procedura che implementa il servizio: le
   funzioni del protocollo RPC ricevono l'argomento e lo
   mettono in un'area di memoria del server passando
   l'indirizzo al server */
u_long * proc5 (a)
u_long *a;
{ static u_long   b;
     b = *a + 100;
     return((u_long *)&b);
}
/* il risultato deve essere in una variabile di tipo statico
   alla terminazione della procedura non deve essere deallocato
   come le variabili automatiche
   la procedura locale sarebbe
         u_long proc5 (a) u_long a;
         { return a + 100;}
*/

main()
{    registerrpc (RPC5, VER5, PRC5, proc5,
                  xdr_u_long, xdr_u_long);
     svc_run();  /* attesa indefinita */
     fprintf(stderr,"Errore: svc_run ritornata!\n");  exit(1);
}
```

Dopo la registrazione della procedura, il processo sta in attesa di chiamate: la primitiva *svc_run()* non termina

la primitiva **registerrpc()** vale per il processo che esegue la **svc_run** che esprime la attesa del server



I Processi Server attendono le richieste

Le procedure registrate con **registerrpc**() sono compatibili con chiamate realizzate da primitive basate su meccanismi di trasporto **UDP** senza connesione
incompatibili con **TCP** a stream

Uso di **processi servitori** che contengono i servizi (anche più di uno) e sono in attesa

Corrispondenza dei NOMI

In realtà, la tabella si basa su una **registrazione fisica**

tripla
{numero_progr, numero_vers, protocollo_di_trasporto}
e
numero di porta

in una tabella detta ***port map***

manca il numero di procedura dato che tutti i servizi RPC condividono lo stesso **gestore di trasporto**

la tabella è strutturata come insieme di elementi

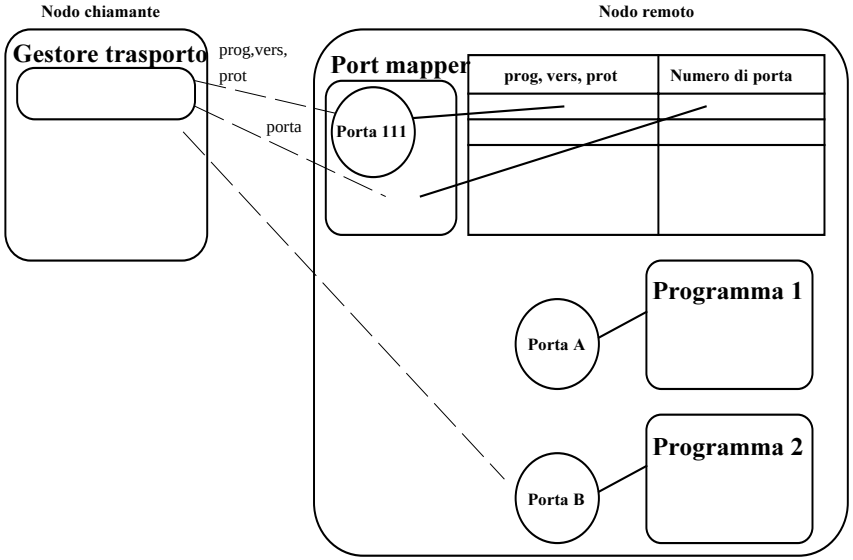
```
struct mapping    {
     unsigned long prog;
     unsigned long vers;
/* uso delle costanti IPPROTO_UDP ed IPPROTO_TCP */
     unsigned int prot;

     unsigned int port;
/* corrispondenza con la porta assegnata */
                    };

/* implementazione a lista dinamica della tabella */
struct *pmaplist { mapping map; pmaplist next;}
```


PORT MAPPER (SERVER)

La gestione della tabella di **port_map** si basa su un **processo unico** per ogni nodo RPC detto

port mapper

lanciato come *demone* (cioè in background)



Il port mapper abilita due gestori di trasporto propri
uno per **UDP** ed
uno per **TCP**
con due socket legate allo stesso numero di porta (111)

il numero di programma e versione del **port mapper**
100000 2

Motivazioni

Per limitare il numero di porte riservate ==>
Un solo processo su una porta riservata (*111*)

Il port mapper identifica il numero di porta associato ad un qualsiasi programma ==>

allocazione dinamica dei servizi sui nodi

Il port mapper registra i **servizi** sul nodo
e offre procedure per ottenere informazioni

- **Inserimento** di un servizio
- **Eliminazione** di un servizio
- **Corrispondenza associazione astratta e porta**
- **Intera lista** di corrispondenza
- **Supporto** all'esecuzione remota

port mapper utilizza a default solo il trasporto **UDP** (*perché?*)

Questi servizi possono essere chiamati da locale o remoto
Per esempio, la PMAPPROC_SET di norma locale
primitive per distinguere la natura locale/remota del servizio

LIMITI

Dal cliente ogni chiamata interroga il port mapper
(poi attua la richiesta)

Dal server gestione diversa

Protocollo di PORT MAPPER

Nome del servizio	**Nro proc**	**Funzione svolta**
PMAPPROC_NULL	0	Nessuna
PMAPPROC_SET (n_prog, n_vers, prot, port)	1	**Inserisce** una **entry** nella tabella: {n_prog, n_vers, prot e nro porta} Operazione **indivisibile**: se l'associazione esiste fallisce
PMAPPROC_UNSET (n_prog, n_vers, prot, port)	2	**Elimina** una **entry** dalla tabella se non esiste, fallisce
PMAPPROC_GETPORT (n_prog, n_vers, prot, dummy)	3	**analisi della tabella** con {n_prog, n_vers, prot} restituisce il **numero di porta**
PMAPPROC_DUMP	4	Restituisce l'**intera lista** che implementa la tabella
PMAPPROC_CALLIT (n_prog, n_vers, n_proc, arguments)	5	**Esecuzione senza** specificare la **porta** {n_prog, n_vers, n_proc, argomenti} ed il risultato è il numero di porta I risultati dell'esecuzione remota sono ritornati

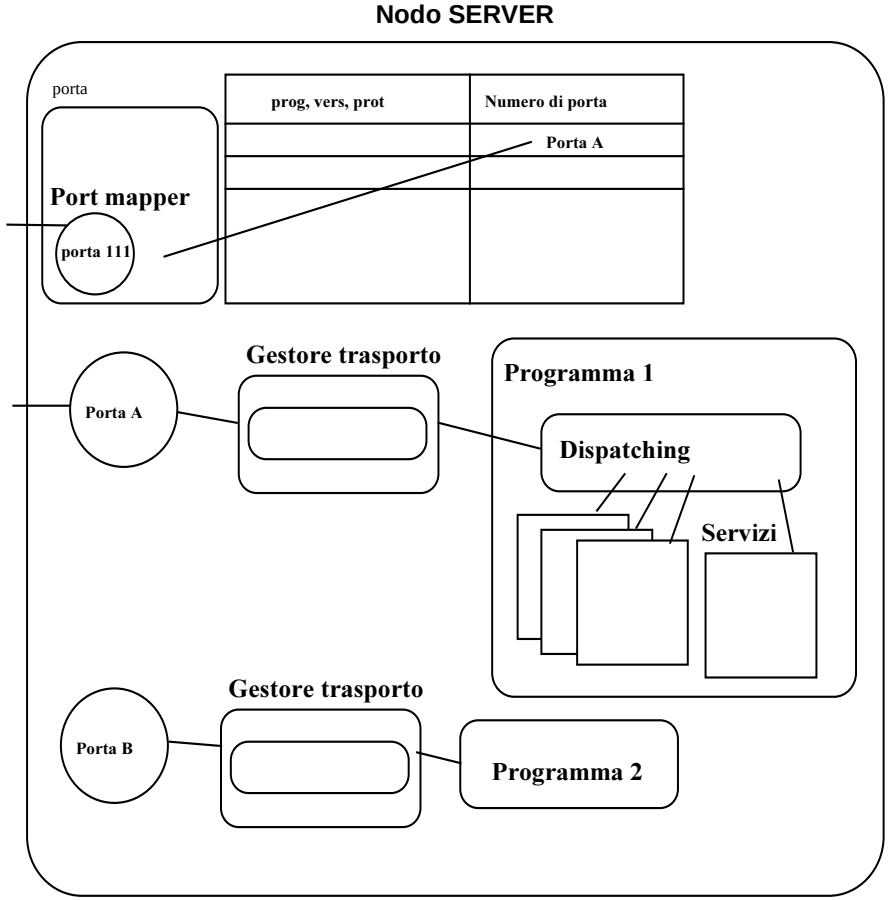
Si veda **rpcinfo -p nomehost**

Servizi remoti (tra i parametri, l'indirizzo Internet remoto)	**Servizi locali (effetto sul port mapper locale)**
struct pmaplist * ***pmap_getmaps*** (addr) struct sockaddr_in *addr; */* nel socket address numero di porta 111 e indirizzo Internet del nodo remoto */*	bool_t ***pmap_set*** (n_prog, n_vers, prot,port) u_long n_prg, n_vrs, prot; u_short port;
u_short ***pmap_getport*** (addr,n_prg ,n_vrs,prot) struct sockaddr_in *addr; u_long n_prg, n_vrs,prot;	bool_t ***pmap_unset*** (n_prg,n_vrs) u_long n_prog,n_vers;
enum clnt_stat ***pmap_rmtcall*** (addr,n_prog,n_vrs, n_prc, xdr_arg, arg, xdr_res, res, timeout, port) struct sockaddr_in *addr; u_long n_prg, n_vrs, n_prc; xdrproc_t xdr_arg; char *arg; char *res; xdrproc_t xdr_res; struct timeval timeout; u_short port;	

pmap_getport() per ottenere la porta a cui inviare RPC
pmap_set() per la registrazione (e **pmap_unset**())
anche possibile la chiamata remota in un unico passo
primitiva **pmap_rmtcall**() **UDP**

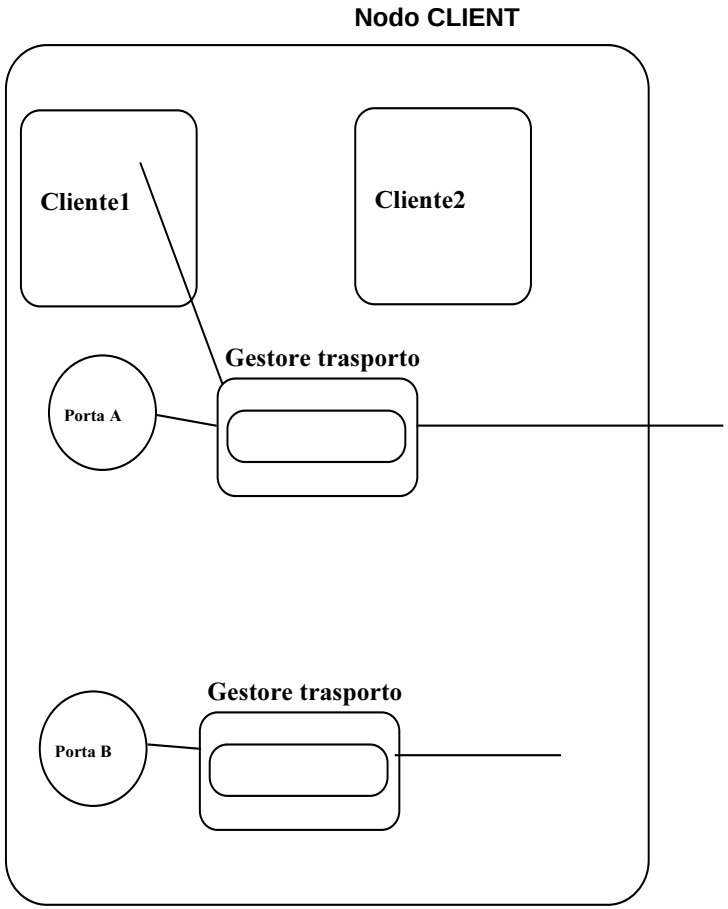
Nodo server

creazione di un gestore di trasporto
per mantenere il collegamento RPC con i clienti



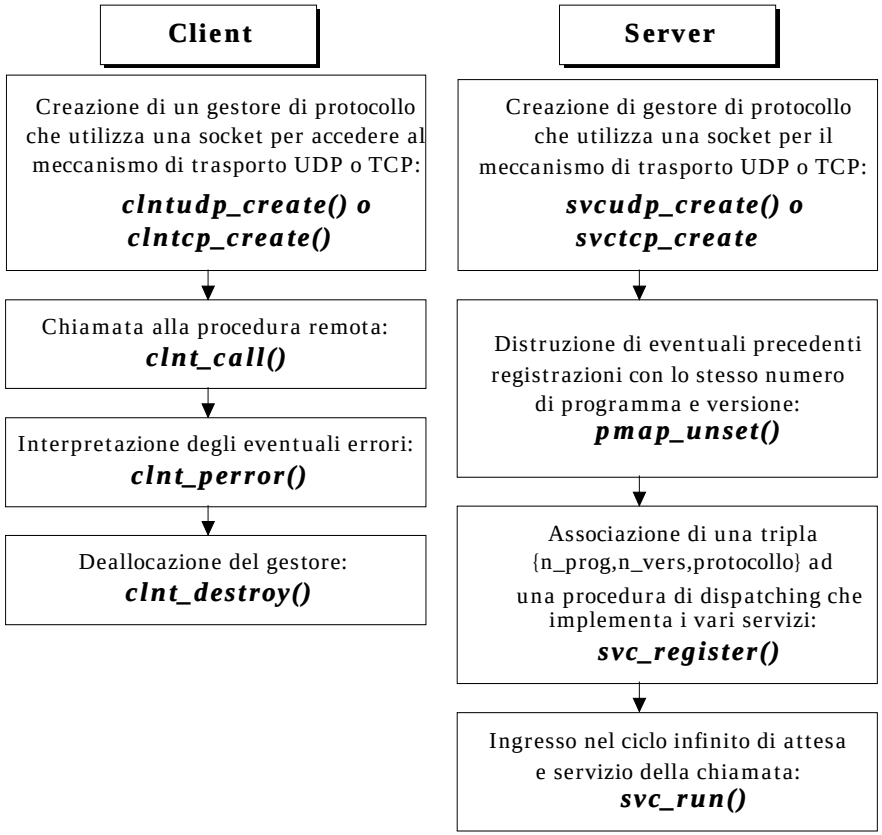
Nodo client

creazione di un gestore di trasporto
per mantenere il collegamento RPC con i clienti



Chiamata remota a basso livello

Chiamata remota tramite funzioni avanzate
per ottenere massima capacità espressiva



Flusso di operazioni per RPC in **gestione avanzata**
La chiamata **svc_register**() e la **svc_run**() possono essere implementate con funzioni di più basso livello

Server
Creazione di un gestore di trasporto

La registerrpc() utilizza la ***svcudp_create()***
per ottenere un gestore UDP
In caso di protocollo **affidabile** si può usare
svctcp_create()

Il **gestore di trasporto** è definito da SVCXPRT

```
typedef struct {
#ifdef KERNEL
    struct socket    * xp_sock;
#else
    int  xp_sock;          /* socket associata */
#endif
    u_short xp_port;    /* numero di porta associato */
    struct xp_ops {
       bool_t    (*xp_recv)();  /* ricezione richieste in arrivo */
       enum xprt_stat (*xp_stat)(); /*  stato del trasporto */
       bool_t    (*xp_getargs)();    /* legge gli argomenti */
       bool_t    (*xp_reply)();      /* invia una risposta */
/* libera la memoria allocata per gli argomenti */
       bool_t    (*xp_freeargs)();
       void      (*xp_destroy)();    /* distrugge la struttura */
    } * xp_ops;
    int  xp_addrlen;    /* lunghezza dell'indirizzo remoto */
    struct sockaddr_in xp_raddr;  /* indirizzo remoto */
    struct opaque_auth xp_verf;    /* controllore risposta */
    caddr_t      xp_p1;              /* privato */
    caddr_t      xp_p2;              /* privato */
} SVCXPRT;
```


Gestore di trasporto

È una ***struttura astratta*** che
- punta alle operazioni sui dati
- riferisce due socket e una porta (locale)
  una per il protocollo di trasporto del server *(xp_sock)*
  una (se richiesta in atto) a cui inviare i risultati della esecuzione remota *(xp_raddr)*

SVCXPRT * **svcudp_create** (sock)
    int sock;

SVCXPRT * **svctcp_create** (sock,
            send_buf_size, recv_buf_size)
    int sock;
    u_int send_buf_size,recv_buf_size;

Se **svcudp_create**()
se sock vale ==> RPC_ANYSOCK
    si crea un datagram socket per i risultati
Altrimenti, il parametro è un socket descriptor (collegata ad uno specifico numero di porta o meno)
    Se la socket associata al gestore non ha un numero di porta e non si tratti di RPC_ANYSOCK, si genera un numero di porta in modo automatico

La **svctcp_create**() funziona in modo analogo
si devono definire le dimensioni dei buffer tramite cui si scambiano i dati

In caso di insuccesso, non si crea il gestore

Server

Procedura di dispatching

La procedura di *dispatching* contiene i riferimenti alle ***implementazioni dei servizi*** di un programma RPC
==> lo stesso gestore e lo stesso protocollo di trasporto

la procedura di dispatching seleziona il servizio da eseguire interpretando il messaggio RPC consegnato dal gestore

```
struct svc_req {
    u_long      rq_prog;        /* numero di programma */
    u_long      rq_vers;        /* versione */
    u_long      rq_proc;        /* servizio richiesto */
    struct opaque_auth rq_cred;    /* credenziali */
    caddr_t     rq_clntcred;    /* credenziali di sola lettura */
    SVCXPRT *   rq_xprt;        /* gestore associato */
};
```

void ***dispatch*** (request, xprt)
    struct svc_req *request;
    SVCXPRT *xprt;

rq_proc identifica il servizio da svolgere
- parametri per l'esecuzione ricavati dalla ***svc_getargs()***
- risposta rispedita tramite la ***svc_sendreply()***
    del gestore di trasporto del protocollo RPC

bool_t ***svc_getargs*** (xprt,inproc,in)
    SVCXPRT *xprt;
    xdrproc_t inproc;
    char *in;

bool_t ***svc_sendreply*** (xprt,outproc,out)
    SVCXPRT *xprt;
    xdrproc_t outproc;
    char *out;

funzioni di formato XDR per argomenti e risultati
    I valori di output (getargs) e di input (sendreply) sono già trasformati in formato locale

Ci sono molte altre funzioni per ottenere informazioni sul gestore di trasporto e fornire informazioni ulteriori

la funzione di registrazione inserisce i servizi nel dispatching Inoltre

- la NULLPROC (numero di procedura 0) verifica se il server è attivo
- controllo della correttezza del numero di procedura, in caso contrario *svcerr_noproc()* gestisce l'errore

SERVER

Associazione

numero programma e **versione**
al **dispatching**

Una procedura di dispatching è associata ad una tripla
{*n_prog, n_vers, protocollo*}

mediante la primitiva *svc_register()*

```
bool_t
 svc_register (xprt, prognum, versnum, dispatch,
                              protocol)
    SVCXPRT *xprt; /* gestore di trasporto */
    u_long prognum,versnum;
    char (*dispatch());
    u_long protocol;
```

xptr == gestore di trasporto
prognum, versnum
== identificazione della procedura
dispatch == puntatore alla procedura di dispatching
protocol == tipo di protocollo

non ci sono indicazioni di tipi XDR
solo all'interno dell'implementazione di ogni servizio

E se si registra due volte la stessa *procedura di **dispatching***?

CLIENT

Creazione di un gestore di trasporto

Il client necessita di un gestore di trasporto per RPC

L'applicazione chiamante utilizza ***clntudp_create()***
per ottenere un gestore UDP

Anche ***clnttcp_create()*** per protocollo affidabile

La callrpc() ottiene un gestore di trasporto
con clntudp_create()
A livello intermedio l'interfaccia RPC si basa su UDP

```
typedef struct {
  AUTH     *cl_auth;    /* autenticazione */
  struct clnt_ops {
     enum clnt_stat (*  cl_call)(); /*  chiamata  di  procedura
remota */
     void     (* cl_abort)();   /* annullamento della chiamata */
     void     (* cl_geterr)();  /* ottiene uno codice d'errore */
     bool_t   (* cl_freeres)(); /* libera lo spazio dei risultati */
     void     (* cl_destroy)(); /* distrugge questa struttura */
     bool_t   (*cl_control)();  /* funzione controllo I/ORPC */
                  } * cl_ops;
     caddr_t cl_private;  /* riempimento privato */
} CLIENT;
```

CLIENT ***clntudp_create***
(addr, prognum, versnum, wait, sockp)
struct sockaddr_in *addr;
u_long prognum,versnum;
struct timeval wait;
int *sockp;
CLIENT ***clnttcp_create***
(addr, prognum, versnum, sockp, sendsz, recvsz)
struct sockaddr_in *addr;
u_long prognum,versnum;
int *sockp;
u_int sendsz,recvsz;

wait == durata dell'intervallo di time-out
sendsz, recsz == dimensione dei buffer

Non ci sono riferimenti espliciti alla socket di trasporto ed al socket address per la richiesta di esecuzione remota

Tra i parametri della clntudp_create()
il valore del ***timeout*** fra le eventuali ritrasmissioni
Se il numero di porta all'interno del socket address remoto vale 0, si lancia un'interrogazione al port mapper per ottenerlo

clnttcp_create() non prevede timeout
definisce la dimensione dei buffer di input e di output

L'interrogazione iniziale causa una connessione
l'accettazione della connessione, consente la RPC

CLIENT
Chiamata della procedura remota

Creato il gestore di trasporto si raggiunge un'entità
{*n_prog, n_vers, protocollo*}
tramite il numero di porta relativo
la procedura di dispatching è già selezionata

la ***clnt_call()*** specifica solo
gestore di trasporto
numero della procedura

enum clnt_stat
clnt_call (clnt, procnum,inproc,in,outproc,out,tout)
CLIENT *clnt;
u_long procnum;
xdrproc_t inproc; /* routine XDR */
char *in;
xdrproc_t outproc; /* routine XDR */
char *out;
struct timeval tout;

client == gestore di trasporto locale
procnum == identificazione della procedura
inproc == tipo di argomenti
outproc == tipo di risultato
Routine XDR per serializzare i dati
in == argomento unico
out == risultato unico
tout == tempo di attesa della risposta

- se UDP, il numero di ritrasmissioni è dato dal rapporto fra questo valore ed il timeout indicato nella clntudp_create()
- se TCP, questo parametro indica il timeout oltre il quale il server è considerato irraggiungibile

Si veda la **clnt_control()** che consente di modificare i parametri per un gestore già creato

Analisi degli eventuali errori

Risultato di **clnt_call()** analizzato con *clnt_perror()*
stampa sullo standard error una stringa contenente un messaggio di errore

void ***clnt_perror*** (clnt,s)
    CLIENT * clnt;
    char * s;

I parametri
clnt == gestore di trasporto
s == stringa di output

Distruzione del gestore di trasporto del client

clnt_destroy() dealloca lo spazio associato al gestore CLIENT, senza chiudere la socket

==> Più gestori possono condividere una stessa socket
void ***clnt_destroy*** (clnt)
    CLIENT *clnt;

Gestione avanzata del protocollo RPC

le primitive avanzate dell'interfaccia RPC
per il controllo di protocolli sottostanti:
    manipolare socket BSD
    manipolazione delle strutture XDR

- ***scelta del protocollo di trasporto***
    possibile creare un gestore che si interfacci a TCP invece UDP ==>
    scambiare messaggi più lunghi di 8 Kbytes o
    cambiare la semantica di comunicazione
- ***accesso avanzato alla tabella delle registrazioni***
    meccanismi di gestione e primitive dirette di manipolazione
- ***allocazione e deallocazione di memoria***
    durante il serializing e deserializing
    non esplicita la gestione della memoria per il trattamento dei dati
- ***autenticazione sul lato server e su quello client***
- ***utilizzo del protocollo RPC in modo avanzato***
    metodologie asincrone e concorrenti
    chiamata broadcast
    inversione di ruoli fra client e server

Un **solo portmapper** riceve richieste e riconosce i servitori da attivare
Ogni richiesta genera un servitore opportuno cui si passano i parametri dell'invocazione
Al completamento il server fornisce la risposta

ESEMPIO

Server: funzione ***rnusers***()
con trasporto protocollo RPC su TCP

non la parte locale al servitore che ricava il numero degli utenti attivi sul nodo

Motivazione: possibilità di invio messaggi senza limiti di dimensioni (non realizzabile con UDP)

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

int nuser();
/* dichiarazione procedura di dispatching invocata per
   ogni richiesta*/

main()
{ SVCXPRT *transp;
/* creazione del gestore TCP con la specifica per generare una
   dimensione ottimale dei buffer */
transp = svctcp_create (RPC_ANYSOCK,0,0);
if (transp==NULL)
 {fprintf((stderr,"Non posso creare un gestore di trasporto.\n");
  exit(1);    }
pmap_unset (RUSERPROG,RUSERVERS);
if (! svc_register (transp,RUSERPROG, RUSERVERS,
          nuser /* dispatching */, IPPROTO_TCP))
     {    fprintf(stderr,"Non posso registrare il servizio rnuser.\n");
          exit(1); }
svc_run ();
fprintf(stderr,"Errore: uscita dal loop infinito di attesa!\n");
}
```



```
nuser(rqstp,transp)  /* procedura di dispatching */
struct svc_req *rqstp;
SVCXPRT *transp;
{      unsigned long nusers;
switch (rqstp->rq_proc) {
/* analizzando il valore del campo rq_proc si ottiene il
   servizio da eseguire */

case NULLPROC:
/*  questa procedura è implementata per consentire la
   semantica della registerrpc() */
if (! svc_sendreply (transp, xdr_void,0))
/* invio della risposta nulla */
  { fprintf(stderr,"Non posso rispondere alla chiamata RPC.\n");
    exit(1);  }
 return;

case RUSERPROC_NUM:
<< qui ci vuole la parte che calcola quanti utenti ci sono
    sul nodo: il risultato in nusers >>
if (! svc_sendreply (transp, xdr_u_long, &nusers)
  {  fprintf(stderr,"Non posso rispondere alla chiamata RPC.\n");
     exit(1); }
 return;

case RUSERPROC_BOOL:
{int bool;   unsigned long nuserquery;
 if (! svc_getargs (transp,xdr_u_long,&nuserquery))
{    /* acquisizione degli argomenti della chiamata remota */
     svcerr_decode(transp);
     return;  }
<< qui ci vuole la parte che calcola quanti utenti sono
    presenti sul nodo e pone il risultato in nusers >>
```

```
    if (nuserquery == nusers)
/* controllo se il numero di utenti ipotizzato dal client è reale
*/
        bool = TRUE;
    else
        bool = FALSE;
    if (! svc_sendreply (transp, xdr_bool, &bool)
{  fprintf(stderr,"Non posso rispondere alla chiamata RPC.\n");
   exit(1);   }
    return;
}

default:
    svcerr_noproc (transp);
    return;
    }
}
```


codice di un processo cliente

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>
#include <sys/socket.h>
#include <time.h>
#include <netdb.h>

main(argc,argv)
  int argc;
  char *argv[];
{ struct hostent *hp;
  struct timeval total_timeout;
  struct sockaddr_in server_addr;
  int sock = RPC_ANYSOCK;
  register CLIENT *client;
  enum clnt_stat clnt_stat;
  unsigned long nusers, nuquery;
  int bool;

if (argc<2) {    fprintf(stderr,"uso:%s nomehost\n",argv[0]);
                 exit(1); }
if ((hp=gethostbyname(argv[1]))==NULL)
{fprintf(stderr,"Non posso avere l'indirizzo di %s\n",argv[1]);
 exit(1);    }
memcpy((caddr_t)&server->addr.sin_addr,
        hp->h_addr,hp->h_length);
server_addr.sin_family=AF_INET;
server_addr.sin_port=0;
```

```
/* il port mapper remoto fornisce il numero di porta */
if ((client= clnttcp_create (&server_addr, RUSERPROG,
                RUSERVERS, 0, 0, &sock))==NULL)
{  clnt_pcreateerror("clnttcp_create");  exit(1); }

/* definisco il timeout totale d'attesa della risposta */
total_timeout.tv_sec=20;
total_timeout.tv_usec=0;
clnt_stat= clnt_call (client, RUSERPROC_NUM, xdr_void, 0,
              xdr_u_long, &nusers, total_timeout);
if (clnt_stat!=RPC_SUCCESS)
  {  clnt_perror (client,"RPC");
     exit(1); }

total_timeout.tv_sec = 10;
nuquery=3;      /* numero di utenti ipotizzato. */
clnt_stat= clnt_call (client,RUSERPROC_BOOL, xdr_u_long,
            &nuquery, xdr_bool, &bool, total_timeout);
if (clnt_stat!=RPC_SUCCESS)
  {    clnt_perror (client,"RPC");
     exit(1); }

/*  si possono fare un numero qualunque di richieste
    usando la stessa connessione
*/

/* deallocazione finale del gestore di trasporto */
clnt_destroy (client);
}
```


Gestione di eventi multipli durante l'attesa

Il server attende le RPC all'interno del ***svc_run()***
Il processo bloccato potrebbe
svolgere altre attività durante l'attesa

Periodico aggiornamento di strutture dati ==>
impiego di un segnale di tipo ALARM per sbloccare la svc_run() ed eseguire il gestore del segnale
Il gestore risponde all'evento ad ogni invocazione

Se si vogliono usare I/O e file descriptor =>
bisogna modificare **svc_run()**
(che è una procedura e non una primitiva)

Per la gestione di eventi di I/O si definisce una funzione da chiamare al posto di **svc_run()** in cui la maschera passata alla primitiva **select()** contenga una serie di file descriptor

la variabile s**vc_fdset** riporta alcuni file descriptor legati alle RPC

svc_getreqset termina solo se i servizi sono stati tutti soddisfatti

Uso delle macro in file <stdio.h> per agire sulle maschere

Se occorre gestire anche scritture ed eventi eccezionali sarà necessario passare altre maschere alla chiamata select()

Esempio di alternativa alla svc_run

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

void svc_run()
{    fd_set readfds;    /* maschera di file descriptor */

for (;;)   /* ciclo idealmente infinito */
{ readfds = svc_fdset; /* maschera file descriptor RPC */
 switch
     (select(FD_SETSIZE,&readfds,NULL,NULL,NULL))
/* FD_SETSIZE è il massimo numero di file descriptor */
  {
 case -1:
     if (errno==EINTR) continue;
/* l'unico caso in cui l'errore viene ignorato è quello di
   interruzione a causa del segnale ALARM
*/
     perror("svc_run: select");return;
case 0:
     break;
default:
     svc_getreqset (&readfds);
/* questa primitiva termina solo quando tutte le socket indicate
   dalla maschera sono state lette e le relative procedure di
   dispatching sono terminate. Al termine si riprende l'attesa
*/
          }
     }
}
```

Si possono anche fare una gestione con time-out della select

Broadcast RPC

Unica richiesta Client-Server

*un **unico** client e più **server** =>***broadcast**

Il pacchetto è disponibile a livello fisico a molti nodi ma un solo nodo (e server) lo considera

> *IP permette di specificare un indirizzo di destinazione visibile ad ogni nodo*
> *il pacchetto usa datagram socket, attraverso UDP (limiti di dimensioni 1400 byte)*

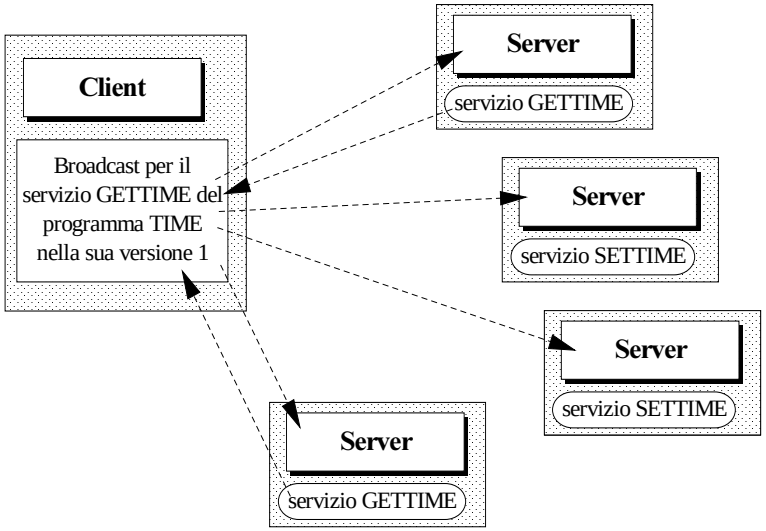
Problemi di sincronizzazione della chiamata remota

Ogni nodo che riceve la richiesta risponde al nodo mittente se prevede il servizio
Si usa il **portmapper** per ottenere la **esecuzione remota**

si realizza una operazione *multicast*

> le risposte ricevute dal client possono addirittura essere un numero non definibile prima della chiamata e giungere in istanti di tempo molto distanti fra loro

Modello broadcast



RPC con indirizzo internet di broadcast e
un unico numero di porta destinataria

Impiego del servizio **PMAPPROC_CALLIT**

Il socket address deve contenere
indirizzo internet di tipo broadcast e
numero di porta 111

protocollo di trasporto di tipo connection-less
procedura PMAPPROC_CALLIT
solo esecuzioni di servizi basati su UDP

Si veda **rpcinfo -b**

Differenze fra RPC e modello broadcast

- le RPC una sola risposta, broadcast **risposte multiple**
- protocollo di trasporto **connectionless** (UDP)
- modello broadcast non gestisce gli **errori** nella chiamata
- i messaggi di chiamata sono inviati ai servizi tramite il programma **port mapper**

Interfaccia RPC ==> modello broadcast
clnt_broadcast()

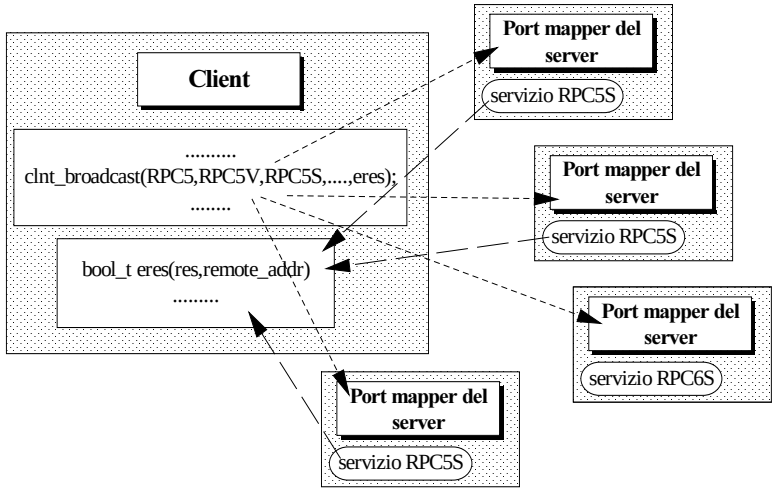
```
#include <rpc/rpc.h>
enum clnt_stat clnt_broadcast (prog, vers, proc, xargs,
            argsp, xresults, resultsp, eachresult)
    u_long      prog, vers, proc;
    xdrproc_t   xargs;  /* tipo argomenti */
    caddr_t     argsp;  /* lunghezza argomenti */
    xdrproc_t   xresults;
    caddr_t     resultsp;
    bool_t (* eachresult)();
```

Ultimo argomento riferimento ad una **funzione** chiamata
ogni volta che viene ricevuto un risultato valido
la funzione deve restituire un booleano
che specifica se il client deve concludere l'attesa

```
bool_t eachresult (resultsp, raddr)
    caddr_t resultsp;  struct sockaddr_in * raddr;
```

da dove arriva il risultato

Implementazione del modo broadcast



Per sapere quale nodo ha risposto
==> indirizzo internet contenuto nel socket address
eachresult() analizza risultati per stabilire se terminare
Con risultato vero il broadcast deve terminare
clnt_broadcast() termina

Chiamata **clnt_broadcast()** richiede un servizio simile a quello di livello intermedio
La funzione **eachres()** attivata dall'arrivo di una generica risposta deve visualizzare: l'indirizzo di rete del nodo server, il numero di porta, l'istante di arrivo del risultato ed il risultato stesso
eachres termina solo se si è già ottenuto il numero desiderato di risposte

Se la funzione eachres() non restituisce vero entro 55 secondi, la clnt_broadcast() restituisce RPC_TIMEDOUT

Esempio

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>
#include <netdb.h>
#include <netinet/in.h>
#define RPCP5 (unsigned long) 0x20000015
#define RPCV5 (unsigned long) 1
#define RPCS5 (unsigned long) 1
/* variabile globale per il numero di risposte attese */
int maxrepl;
bool_t eachres (); char * ctime ();
/* dichiarazione delle funzioni di gestione e dell'orario */

main(argc,argv) int argc;  char *argv[];
{    enum clnt_stat clnt_stat;
     unsigned long b;
     unsigned long a=3;
     long timevar; /* valore di ritorno dalla funzione time */
if (argc!=2)
{ printf("Uso: %s <risposte attese>\n",argv[0]); exit(1);    }
maxrepl=atoi(argv[1]); time(&timevar);
printf("Inizio broacast alle %s",ctime(&timevar));
clnt_stat = clnt_broadcast (RPC5P, RPC5V, RPC5S,
          xdr_u_long, &a, xdr_u_long, &b, eachres);
if ( clnt_stat!=RPC_SUCCESS)
{    time(&timevar);
     fprintf(stderr,"Errore ");
     fprintf(stderr," nella broadcast RPC alle %s",
               ctime(&timevar));
     clnt_perrno (clnt_stat); exit(1); }
printf("Risultato: %d\n",b); exit(0); }
```

```
char *inet_ntoa();
/* trasforma da intero IP in ascii in notazione a.b.c.d */

bool_t eachres (resultsp,raddr)
unsigned long *resultsp;
struct sockaddr_in *raddr;
{    struct hostent *hp;
      char *hostname = malloc (120);
      static int nreply=0;
      long timevar;

nreply++;
hp = gethostbyaddr ((char *) &raddr-> sin_addr,
         sizeof (struct in_addr), raddr-> sin_family);
if (hp==NULL) hostname = inet_ntoa ((*raddr).sin_addr);
else              hostname = hp->h_name;

time(&timevar);
fprintf(stderr,"Risposta %d (risultato %d). Host remoto \
%s porta %d. %s", nreply, *resultsp, hostname,
    ntohs((*raddr).sin_port), ctime(&timevar));

if (nreply == maxrepl)
    return(TRUE);/* termina il broadcasting */
else
    return(FALSE);
}
```


Modalità asincrona

RPC client sincrono con il server

Possiamo intervenire sul cliente

- il cliente usa TCP
- se il cliente specifica un time-out nullo può continuare immediatamente la esecuzione
- il servitore non predeve risposta

specifica di un **time-out nullo**
nella **clnt_call**
(clnt, procnum,inproc,in,outproc,out,tout)
per ogni chiamata

nella **clnt_control** (clnt,CLSET_TIMEOUT, timeout)
CLIENT *clnt;
opzione di time out;
struct timeval timeout;

timeout. tv_sec = timeout. tv_nsec = 0;
nessun tempo di attesa della risposta

Il servitore non deve inviare alcuna risposta
Nella procedura di **risposta** si deve dichiarare
xdr-void come funzione XDR e 0 come argomento

IMPORTANZA dell'uso di trasporto affidabile per evitare di perdere messaggi

Modalità asincrona batch

RPC client sincrono con il server
tutte le richieste di servizio vengono poste in *pipeline*
senza bloccare il processo che le genera

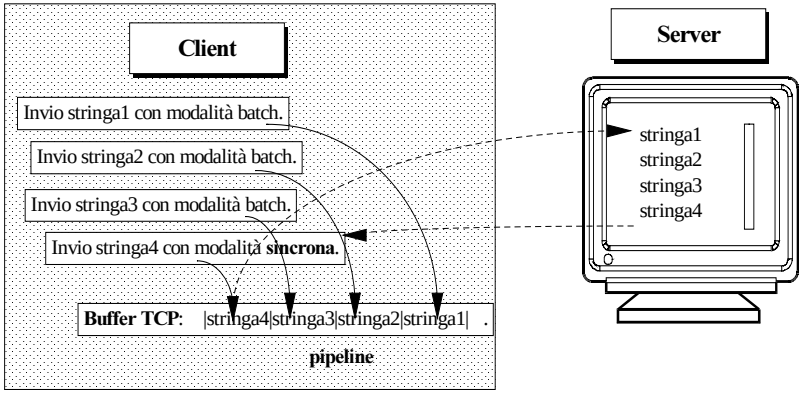
sistema Mercury
modalità *asincrona* prende il nome di *batch*
protocollo TCP

L'invio della pipeline avviene con
una sola chiamata **write**()
ATTESA se
pipe piena oppure
una RPC che attende risultati

modalità batch
- *ogni chiamata non richiede risposta e ogni servizio non invia risultati*
- *la pipeline delle chiamate è trasportata con un protocollo affidabile come TCP*

Implementativamente:
- impiego del protocollo TCP
- valore nullo del timeout nella primitiva clnt_call()
- i due parametri del cliente:
  risultato NULL e funzione XDR xdr_void in clnt_call()
- manca la chiamata svc_sendreply() al termine del servizio asincrono

Esempio
una serie di chiamate asincrone per la stampa di stringhe sul nodo remoto: si termina con una chiamata sincrona che svuota la pipeline



/* CLIENT */

```
#define PROG (unsigned long) 0x20000020
#define VERS (unsigned long) 1
#define PRINTSTRING (unsigned long) 1
#define PRINTSTRING_BATCHED (unsigned long) 2
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/socket.h>
#include <time.h>
#include <netdb.h>
main(argc,argv)
int argc;  char *argv[];
{     struct hostent *hp;
       struct timeval total_timeout;
       struct sockaddr_in server_addr;
       int sock = RPC_ANYSOCK;
       register CLIENT *client;
       enum clnt_stat clnt_stat;
       char buf[BUFSIZ], *s=buf;
```

```
if (argc<2) {fprintf(stderr,"uso: %s hostname\n",argv[0]);
             exit(1); }
if ((hp=gethostbyname(argv[1]))==NULL)
{    fprintf(stderr,"non ho informazioni su %s\n", argv[1]);
     exit(1); }
memcpy((caddr_t)&server_addr.sin_addr, hp->h_addr,
                                hp->h_length);
server_addr.sin_family=AF_INET; server_addr.sin_port=0;

if ((client= clnttcp_create (&server_addr, PROG,VERS,
                  &sock,0,0))==NULL) /*  gestore  TCP */
   {clnt_pcreateerror ("clnttcp_create\n");  exit(1);    }
/* il timeout sulla risposta RPC è posto a zero */
total_timeout.tv_sec=0;     total_timeout.tv_usec=0;
/* ciclo di lettura di stringhe da tastiera per passarle al nodo
  remoto: si entra nella  pipeline e verranno spedite solo allo
  svuotamento */
while (scanf("%s",s)!=EOF)
{ clnt_stat = clnt_call (client, PRINTSTRING_BATCHED,
     xdr_wrapstring,&s,xdr_void, NULL,total_timeout);
/* risultato e funzione XDR a NULL */
     if (clnt_stat != RPC_SUCCESS)
     {    clnt_perror(client,"RPC asincrona");  exit(1); }
}
/* si svuota la pipeline con una normale RPC sincrona */
total_timeout.tv_sec=20;
clnt_stat=clnt_call (client,NULLPROC, xdr_void, NULL,
          xdr_void, NULL, total_timeout);
if (clnt_stat != RPC_SUCCESS)
     { clnt_perror(client,"rpc"); exit(1); }
clnt_destroy(client);
}
```


/* SERVER */

```
#define PROG (unsigned long) 0x20000020
#define VERS (unsigned long) 1
#define PRINTSTRING (unsigned long) 1
#define PRINTSTRING_BATCHED (unsigned long) 2
#include <stdio.h>
#include <rpc/rpc.h>

void printdispatch();
main()
{  SVCXPRT *transp;

transp= svctcp_create (RPC_ANYSOCK,0,0);
/* il gestore è TCP */
if (transp==NULL)
     {fprintf(stderr,"cannot create an RPC server\n");
      exit(1);}
pmap_unset(PROG,VERS);
if (!svc_register(transp, PROG, VERS, printdispatch,
     IPPROTO_TCP))
{    fprintf(stderr,"cannot register PRINT service\n");
     exit(1); }
svc_run();
fprintf(stderr,"uscita dal ciclo di attesa di richieste!\n");
}

void printdispatch  (rqstp, transp)
     struct svc_req *rqstp;  SVCXPRT *transp;
{ char *s=NULL;
switch (rqstp->rq_proc) {
case NULLPROC:
     if (!svc_sendreply(transp, xdr_void,0))
     {  fprintf(stderr,"non posso rispondere.\n"); exit(1);    }
      fprintf(stderr,"Fine!\n"); return;
```

```
case PRINTSTRING:
     if (!svc_getargs (transp, xdr_wrapstring,&s))
     {  fprintf(stderr,"non posso decodificare gli argomenti.\n");

          svcerr_decode (transp);
          break;    }
     fprintf(stderr,"%s\n",s);
     if (!svc_sendreply (transp, xdr_void, NULL))
       {/* questo è un servizio sincrono */
          fprintf(stderr,"non posso rispondere.\n");
          exit(1);   }
     break;

case PRINTSTRING_BATCHED:
     if (!svc_getargs (transp, xdr_wrapstring, &s))
     {  fprintf(stderr,"non posso decodificare gli argomenti.\n");
          break;    }
     fprintf(stderr,"%s\n",s);
/* questo è un servizio asincrono: non c'è svc_sendreply() */
     break;
default:
     svcerr_noproc (transp);
     return;
     }
svc_freeargs (transp,xdr_wrapstring,&s);
/* funzione di basso livello per deallocare gli argomenti
acquisiti con la chiamata svc_getargs() */
}
```

L'esecuzione dei due programmi permette all'utente di digitare su di un terminale stringhe e, solo all'EOF compaiono a video sul terminale remoto

Lo svuotamento (flush) della pipeline avviene solo al termine della lettura da tastiera

Concorrenza nel modello RPC

realizzazione di un **modello multi-server**
di un **modello multi-client**

Esecuzione di processi concorrenti
nell'implementare applicazioni basate su RPC

Ad esempio
progetto di un *file system distribuito*
allocazione fisica dei file su nodi remoti
con trasparenza all'utente

Le operazioni del file system su un file remoto
==> necessitano di un supporto di comunicazione RPC

Assumiamo un programma RPC con questi servizi
Richieste contemporanee da clienti del file system

Eventi per servire una richiesta:

- il server associa una procedura di dispatching ad un insieme di triple {*programma, versione, protocollo*}
- il server attende richieste di servizio usando **svc_run()**: attesa sulle socket indicate da **svc_fdset** e quella associata al programma RPC
- in caso di più richieste, la socket associata segnala la presenza di messaggi e si attiva la **svc_getreqset()** Più chiamate al dispatching
- al termine di **svc_getreqset()** si torna nella select()

Concorrenza dei servizi

Il dispatching può passare immediatamente alla attesa di nuovi messaggi, lasciando un processo concorrente per l'esecuzione del servizio.

La creazione di un processo concorrente per ogni richiesta è molto costosa

==>

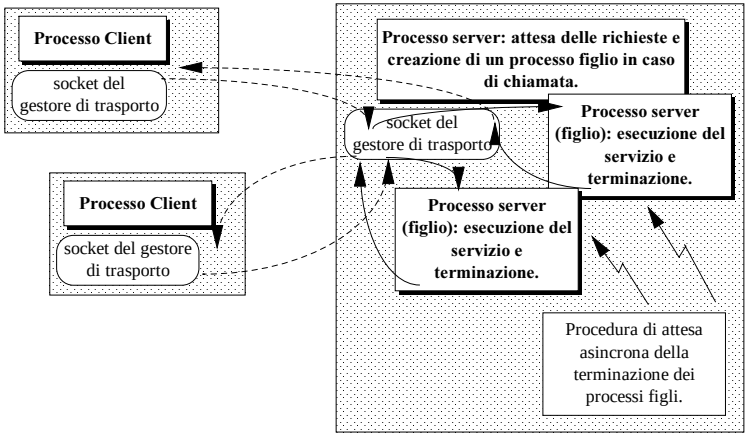
possibilità di creare un certo numero di processi pronti per l'esecuzione del servizio

Generazione dei processi concorrenti ad ogni richiesta

Uso della *fork()*

processo figlio che esegue il servizio

il processo padre esce dalla procedura di dispatching e rientra ad aspettare le RPC



Il processo padre non attende la terminazione del figlio

per deallocare l'area dati del figlio e registrarne la terminazione

==>

ad ogni terminazione di una procedura di servizio, il processo figlio diventa *zombie*

wait3() con comportamento **asincrono** non bloccante

per tutti i processi generati, legge lo stato di terminazione e dealloca

#include <sys/wait.h>
pid_t **wait3** (stat_loc, options, reserved)
int *stat_loc; int options; int *reserved;

stat_loc == stato di ritorno del figlio
options == opzioni

opzioni di funzionamento: WNOHANG
primitiva non bloccante

Esempio:

wait3 (&status, ***WNOHANG***, NULL)
/* primitiva senza attesa non bloccante */

```
#include <stdio.h>
#include <dirent.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <values.h>
#include <rpc/rpc.h>
#include <signal.h>
#include <sys/wait.h>

#include "sr.h"
/* deve contenere le definizioni dei tipi di dati per argomenti e
   risultati dei servizi e la definizione delle costanti per il
   numero di programma, versione e procedure */

void un_register_prog (signo)
int signo;
/* gestore dei segnali SIGHUP, SIGINT, SIGQUIT e
   SIGTERM: si occupa di cancellare la registrazione alla
   terminazione del processo server per evitare che le chiamate
   giungano senza che vi sia un processo che le ascolti e le
   serva */
{    pmap_unset(RCDFS, RCDFSV);
     exit(1); }

/* dichiarazione della procedura di dispatching */
static void rcdfs_1();
/* dichiarazione della funzione che si occupa dell'attesa della
   terminazione dei processi figli generati per le richieste */
int reaper ();
```



```
main()
{ SVCXPRT *transp;  /* gestore di trasporto */
pmap_unset (RCDFS, RCDFSV);
/* aggancio dei gestori dei segnali */
(void) signal(SIGHUP, un_register_prog);
(void) signal(SIGINT, un_register_prog);
(void) signal(SIGQUIT, un_register_prog);
(void) signal(SIGTERM, un_register_prog);
(void) signal(SIGCHLD, reaper);
/* il segnale SIGCHLD alla terminazione di un figlio */
transp = svctcp_create (RPC_ANYSOCK, 0, 0);
if (transp == NULL)
/* il protocollo scelto è TCP per maggiori garanzie sul
   trasporto di dati particolari come i file */
     {fprintf(stderr, "non posso creare un gestore TCP.\n");
      exit(1);}
if (!svc_register (transp, RCDFS, RCDFSV, rcdfs_1,
               IPPROTO_TCP))
     {fprintf(stderr, "non posso registrare il programma.\n");
      exit(1);}
svc_run();
fprintf(stderr, "uscita dalla svc_run() \n");
exit(1);
}

reaper()
{    int status;
     while( wait3 (&status, WNOHANG, NULL) > 0) {
          fprintf(stderr,"rilevato uno zombie. \n");
          continue;    }
}
```

```
static void  rcdfs_1 (rqstp, transp)
struct svc_req *rqstp;  SVCXPRT *transp;
/* le definizioni di questi argomenti riguardano l'aspetto
   implementativo dei servizi */
{union {     int ping_1_arg;
             r_arg read_1_arg;
             w_arg write_1_arg;
             char *stat_1_arg;
             char *readdir_1_arg;
             mk_arg mkdir_1_arg;
             rm_arg remove_1_arg;
          } argument;
     char * result;
     bool_t (* xdr_argument )(), (* xdr_result )();
/* puntatori per gli argomenti e risultati e le funzioni XDR
   impiegate per argomenti e risultati */
     char *(* local )();
/* puntatore alla procedura che implementa il servizio
   selezionato */
     int pid;
switch (rqstp->rq_proc) {
     case NULLPROC:
          svc_sendreply (transp, xdr_void, NULL);
          return;
     case PING:
          xdr_argument = xdr_int; xdr_result = xdr_int;
/* ad ogni entry della switch() vengono assegnate le due
   funzioni XDR da impiegare e la procedura che implementa il
   servizio */
          local = (char *(*)()) ping_1; break;
```



```
     case READ:
          xdr_argument = xdr_r_arg; xdr_result = xdr_r_res;
          local = (char *(*)()) read_1;  break;
     case WRITE:
          xdr_argument = xdr_w_arg; xdr_result = xdr_w_res;
          local = (char *(*)()) write_1;  break;
     case STAT:
          xdr_argument= xdr_wrapstring; xdr_result = xdr_s_res;
          local = (char *(*)()) stat_1; break;
     case READDIR:
          xdr_argument = xdr_wrapstring;
          xdr_result = xdr_readdir_res;
          local = (char *(*)()) readdir_1; break;
     case MKDIR:
          xdr_argument = xdr_mk_arg; xdr_result = xdr_int;
          local = (char *(*)()) mkdir_1; break;
     case REMOVE:
          xdr_argument = xdr_rm_arg; xdr_result = xdr_int;
          local = (char *(*)()) remove_1; break;
     default:
          svcerr_noproc(transp); return;
     }

/*   per ogni operazione possibile, si inseriscono le funzioni di
     trasformazione corrette, e si prepara una variabile per la
     generica invocazione, per trattare tutte le operazioni in modo
     unico e parametrico
*/
```

```
/* lettura degli argomenti dal messaggio */
     if (!svc_getargs (transp, xdr_argument,&argument))
     {    svcerr_decode(transp);
          return;  }
     if((pid= fork ())<0)
/* creazione di un processo figlio */
          {fprintf(stderr, "fork() fallita.\n");
           exit(100); }
     if(pid!=0) {          reaper();
                           return;  }
/* padre: prima di tornare alla svc_getreqset() esamina
   l'eventuale esistenza di zombie generati con questa RPC o
   presenti per RPC precedenti */

     result = (* local )(&argument, rqstp);
/* figlio: viene richiamata la procedura richiesta e, se essa
   termina correttamente, vengono spediti i  risultati al
   processo client */
     if (result != NULL && !
/* primitiva per inoltrare i risultati */
          svc_sendreply (transp, xdr_result, result))
          {    svcerr_systemerr (transp);      }
     if (! svc_freeargs (transp, xdr_argument, &argument))
/* dealloca gli argomenti */
          {    fprintf(stderr, "impossibile liberare gli args\n");
               exit(1); }
     exit(0);
}
/* il processo figlio termina la esecuzione:  rimane in stato
   zombie fino a quando il padre non se ne accorge */
```

RICORDARSI DI FARE LA SVC-FREEARGS

Notare

fork() dopo la creazione del gestore di trasporto

condivisione ***dei file descriptor*** fra padre e figlio
e **del gestore** fra padre e figlio

PROBLEMA?

condivisione fra figli

dipende dal protocollo di trasporto

UDP ==> una sola socket

TCP ==> ad ogni messaggio una socket connessa al client, cioè nessuna condivisione fra figli diversi

Il **gestore**

con UDP, conoscendo la socket del chiamante, reinvia alla socket relativa

è importante che arrivi un solo messaggio

con TCP utilizza la connessione del processo

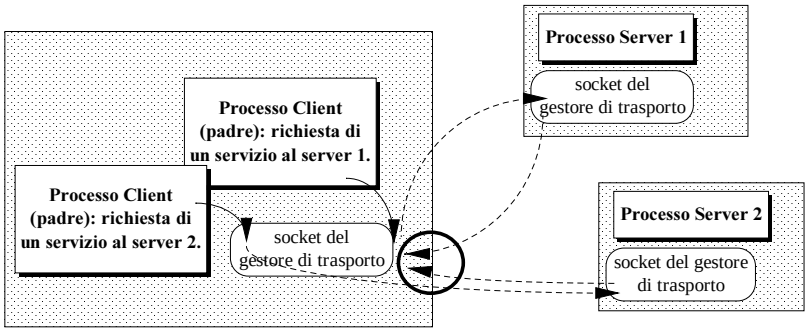
è importante che si crei una connessione per ogni attività del server

Client concorrente più RPC

2 azioni
Creazione del *gestore di trasporto* vs *fork()*

A) fork dopo avere creato il gestore ==>
problemi nella condivisione della socket
Non la spedizione RPC
ma la ricezione dei risultati
CHI garantisce la corretta identità del ricevente?

più processi bloccati in attesa sulla stessa socket
Il primo processo attivo lo ottiene ==>
non si può garantire la corretta destinazione



B) Creazione del gestore di trasporto nei singoli processi

Nel funzionamento reale si può perdere
il segnale SIGCHLD
zombie fino alla RPC successiva poi reaper()

Modalità asincrona di ritorno: CALL-BACK
inversione dei ruoli fra client e server

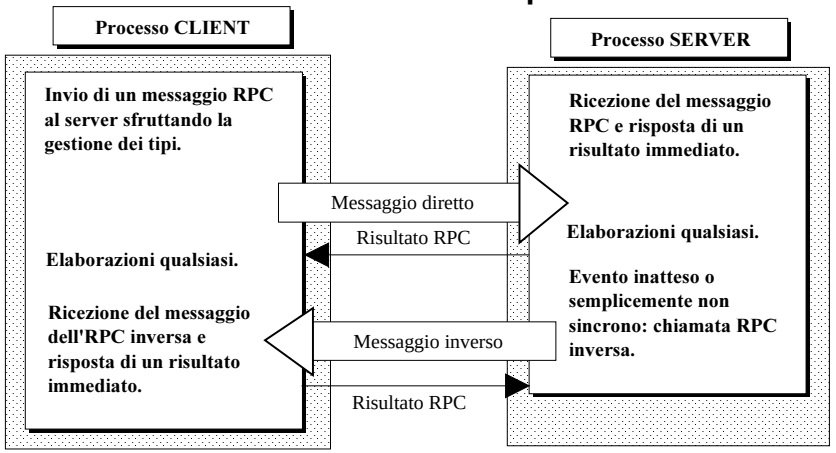
In RPC, i ruoli cliente e servitore ben definiti

alcuni servizi remoti devono invertire il ruolo
servitore diventa richiedente per il cliente
Nella terminologia RPC, si chiama *callback*

Variazione a livello applicativo

callback, anche se RPC iniziale sincrona
==> *asincronismo*

Meccanismo asincrono di tipo callback



Per implementare un meccanismo callback occorre un numero di programma per la RPC di ritorno
==> generato dinamicamente nell'intervallo dei numeri di programma transient (40000000h - 5fffffffh).

Primitiva *gettransient()*

Ricerca di un numero di programma libero da usare
fatto con una procedura che trova un numero libero di programma (**dinamico**)

u_long ***gettransient*** (protocol, vers, sockp)
    int protocol;
    u_long vers;
    int *sockp;

protocol == protocollo utilizzato
vers == versione
sockp == RPC_ANYSOCK, si crea una nuova socket e socket descriptor in *sockp

    risultato è il numero di programma registrato

implementazione di gettransient() per RPC

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

u_long gettransient (proto,vers,sockp)
    int proto;
    u_long vers;
    int *sockp;
{
    static u_long prognum = 0x40000000;
    int s,len,socktype;
    struct sockaddr_in addr;

switch (proto) {
    case IPPROTO_UDP:
        socktype = SOCK_DGRAM;
        break;
    case IPPROTO_TCP:
        socktype = SOCK_STREAM;
        break;
    default:
        fprintf(stderr,"unknown protocol type\n");
        return(0);
                }
```

```
if (*sockp==RPC_ANYSOCK) {
     if ((s= socket (AF_INET, socktype, 0))<0) {
          perror("socket");
          return(0);
          }
     *sockp=s;
     }
else
     s=*sockp;
addr.sin_addr.s_addr=0;
addr.sin_family=AF_INET;
addr.sin_port=0; /* uso di porta attribuita dal sistema
                    altrimenti deve essere nota */
len=sizeof(addr);

/* la socket può essere già collegata: nessuna analisi d'errore
   */
(void) bind (s,&addr,len);
if (getsockname(s,&addr,&len)<0) {
     perror("getsockname");
     return(0);
     }
while(!pmap_set(prognum++,vers,proto,addr.sin_port))
     continue;
/* ciclo di ricerca del numero di programma libero */
return(prognum-1);
}
```


Esempio applicativo di callback

asincronismo legato al segnale ALARM generato dopo 30 secondi dall'avvio del server.

/* * CLIENT */

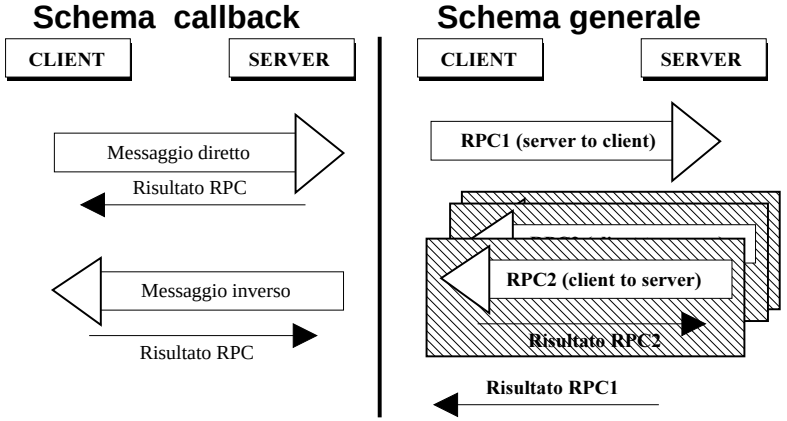
```
#define EXAMPLEPROG (unsigned long) 0x20000016
#define EXAMPLEVERS (unsigned long) 1
#define EXAMPLEPROC_CALLBACK (unsigned long) 1
#include <stdio.h>
#include <rpc/rpc.h>

int callback(); u_long gettransient(),
/* dichiarazione della procedura di dispatching e della
   gettransient */
u_long x;
char hostname[256];

main(argc,argv)
int argc;  char *argv[];
{    int ans,s;    SVCXPRT *xprt;
gethostname(hostname,sizeof(hostname));
s = RPC_ANYSOCK;
x = gettransient (IPPROTO_UDP,1, &s);
/* ricava il numero di programma dinamico */
fprintf(stderr,"generato numero di programma %ld\n",x);
if ((xprt=svcudp_create(s))==NULL) {
     fprintf(stderr,"svcudp_create fallita.\n");
     exit(1); }
(void) svc_register (xprt,x,1,callback,0);
/* registrazione del programma chiamato dal server */
ans= callrpc (hostname, EXAMPLEPROG, EXAMPLEVERS,
     EXAMPLEPROC_CALLBACK, xdr_int, &x, xdr_void,0);
```

```
if (ans != RPC_SUCCESS) {    fprintf(stderr,"chiamata: ");
                          clnt_perrno(ans);  fprintf(stderr,"\n");   }
/* la chiamata RPC d'andata comunica al server il numero di
   programma  ottenuto
   alto livello: non si crea il gestore di trasporto */
svc_run();
printf(stderr,"Errore: svc_run uscita dal ciclo infinito\n");
}


callback (rqstp,transp) /* dispatching del client */
register struct svc_req *rqstp;
register SVCXPRT *transp;
{
switch (rqstp->rq_proc) {
     case NULLPROC:
          if (!svc_sendreply (transp, xdr_void, 0)) {
               fprintf(stderr,"errore:callback\n");exit(1); }
          pmap_unset(x,1);     exit(0);
     case 1:
/* procedura chiamata dal server */
          if (! svc_getargs (transp, xdr_void, 0))
               { svcerr_decode (transp);exit(1);}
          fprintf(stderr,"il client ha ricevuto la callback\n");
          if (!svc_sendreply (transp,xdr_void,0))
          {    fprintf(stderr,"errore:callback\n");
               exit(1); }
}         }
```



```
/*  SERVER */
#define EXAMPLEPROG (unsigned long) 0x20000016
#define EXAMPLEVERS (unsigned long) 1
#define EXAMPLEPROC_CALLBACK (unsigned long) 1
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/signal.h>

char hostname[256];
/* dichiarazione della procedura di servizio della RPC  */
char *getnewprog ();
/* gestore del segnale ALARM: invia la RPC di ritorno */
int docallback ();
u_long pnum=0;
/* contenitore del numero dinamico del client */

main(argc,argv)
int argc;  char *argv[]; /* argomento il nome dell'host cliente */
{ if (argc != 2) exit (1);
  else strcpy (hostname,argv[1]);
 registerrpc (EXAMPLEPROG, EXAMPLEVERS,
                   EXAMPLEPROC_CALLBACK,
                   getnewprog, xdr_int,xdr_void);
fprintf(stderr,"entrata nella svc_run\n");
/* non è necessario lavorare a basso livello
   si utilizza la registerrpc(): questo semplifica la procedura di
   servizio più semplice di quella di dispatching. */
signal(SIGALRM,docallback);
alarm(30);
/*  dopo 30 secondi parte la RPC di ritorno al cliente che deve
   avere registrato il proprio numero */
svc_run ();
fprintf(stderr,"Errore:svc_run ritornata dal ciclo infinito\n");}
```

```
char * getnewprog (pnump)
/* servizio chiamato dal client */
u_long *pnump;
{    pnum = *pnump;
     return(NULL); }

docallback()
/* gestore di SIGALRM per la RPC di ritorno, ossia la callback
   */
{    int ans;
ans = callrpc (hostname, pnum,1,1, xdr_void, 0, xdr_void ,0);
if (ans != 0) {   fprintf(stderr,"server: ");
                  clnt_perrno (ans);
/* anche qui non si opera a livello basso per semplificare la
   chiamata: infatti, non si crea il gestore di trasporto */
                  fprintf(stderr,"\n");
               }
else
     fprintf(stderr,"Invio callback al client.\n");
}
```



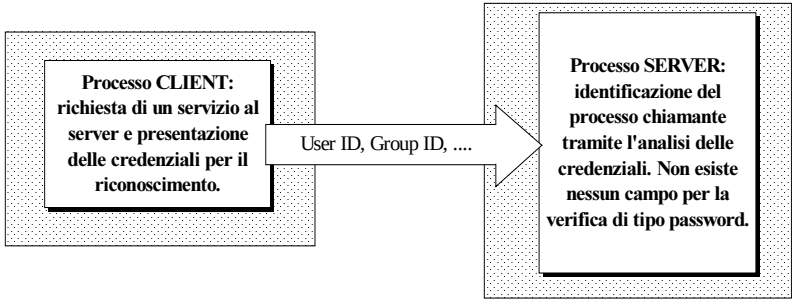
Autenticazioni

Sicurezza nell'accesso ai dati
- Nel messaggio di protocollo RPC
- uno spazio dedicato alle informazioni per autenticare il client presso il server e viceversa

default AUTH_NULL
- ==> mancanza di dati per l'identificazione

AUTH_UNIX si spedisce l'identificazione del mittente, ma non si verificano le credenziali

Passaggio credenziali di un client durante una RPC



L'informazione del tipo di autenticazione scelta per RPC all'interno del gestore di trasporto del processo client
Campo *cl_auth* con un riferimento al
gestore delle autenticazioni

Funzione **clntudp/tcp_create()** inizializza

autenticazione AUTH_NULL

clnt->cl_auth = **authnone_create**();

autenticazione AUTH_UNIX

clnt *CLIENT;
clnt=**clntudp/tcp_create**
(address, prog,vers, wait, sockp);
clnt->cl_auth = **authunix_create_default**();

ogni chiamata RPC nel messaggio di richiesta,
propone le credenziali:
struct **authunix_parms** {
u_long aup_time; /* *orario di creazione delle credenziali* */
char *aup_machname; /* *host mittente* */
int aup_uid; /* *User ID del mittente* */
int aup_gid; /* *Group ID del mittente* */
u_int aup_len;
/* *numero elementi del campo successivo* */
int *aup_gids;
/* *vettore di gruppi a cui il mittente appartiene* */
}

Il contenuto dei campi assegnato automaticamente dalla funzione ***authunix_create_default***()
Per inserimento autonomo ==>
primitiva *authunix_create()*
AUTH * **authunix_create**(host,uid,gid,len,aup_gids)
char *host;
int uid,gid,len,*aup_gids;

risultato è un puntatore ad AUTH, tipo rappresentato dal campo cl_auth nel gestore di trasporto

Necessità di deallocare area occupata

auth_destroy()
auth_destroy (clnt->cl_auth);
che restituisce un puntatore ad AUTH

due campi relativi alle credenziali in svc_req
rq_cred, di tipo *struct opaque_auth*, e
rq_clntcred, di tipo *caddr_t*

struct **opaque_auth** {
enum_t oa_flavor; /* *stile delle credenziali* */
caddr_t oa_base;
u_int oa_lenght;
};

oa_flavor == formato delle credenziali del mittente RPC

RPC, garantisce due condizioni

- il campo rq_cred fornisce l'indicazione del tipo credenziali nel sottocampo oa_flavor
- il campo rq_clntcred punta ad una struttura dati con le credenziali nel formato ricavato come sopra
  Se il formato è UNIX, il puntatore può subire
  un casting a authunix_parms
  Se NULL, il formato non tenuto in conto

Esempio di codice da inserire in una procedura di dispatching per autenticare il mittente con un User ID specifico

```
dispatch (rqstp,transp)
     struct svc_req *rqstp;  SVCXPRT *transp;
{    struct authunix_parms *unix_cred;
     int uid;
/* la NULLPROC può non essere condizionata alle credenziali,
   dunque la si può implementare a questo punto */
switch (rqstp->rq_cred.oa_flavor) {
     case AUTH_UNIX:
/* esegue un casting di tipo dato per il formato corrispondente
   */
          unix_cred =  (struct authunix_parms *)
                              rqstp->rq_clntcred;
          uid = unix_cred->aup_uid;
          break;
     case AUTH_NULL:
     default:
          svcerr_weakauth (transp);
          return;                          }
/* questa funzione fornisce un messaggio di errore per indicare
   la mancanza di credenziali o il formato errato */
switch (rqstp->rq_proc) {
     case WRITE:
     if (uid == 16 ) {     /* implementazione del servizio */  }
          else {    svcerr_systemerr (transp);
                    return;  }
/* segnalazione di un tipo d'errore non appartenente al
   protocollo RPC: un errore di sistema */
/* la procedura continua con l'implementazione di altri servizi.
   */
}
```

RPC non controlla accesso ai dati

Generazione Automatica delle RPC

Maggiore astrazione dell'applicazione ==>
Uso di procedure stub

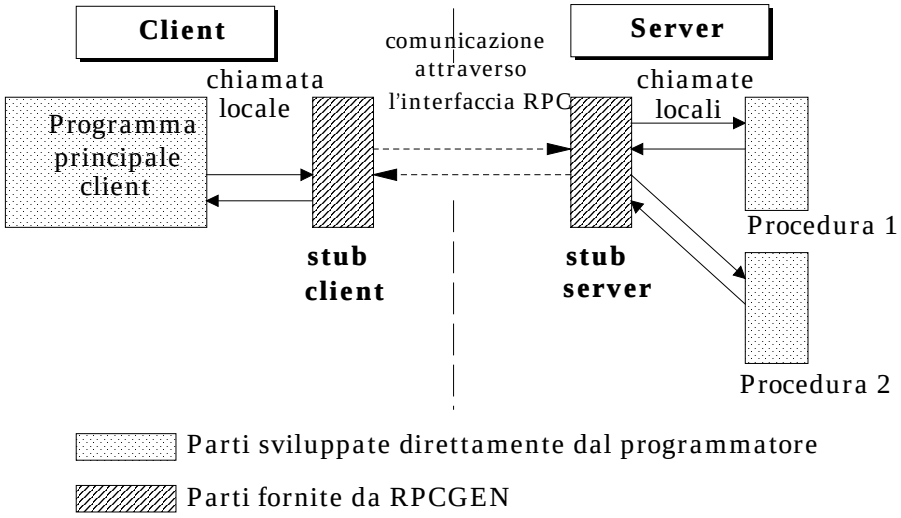
Remote Procedure Call Generator (RPCGEN)

compilatore di protocollo RPC genera procedure stub in modo automatico

RPCGEN processa

un insieme di costrutti descrittivi per tipi di dati e per le procedure remote

==> *linguaggio RPC*



moduli prodotti

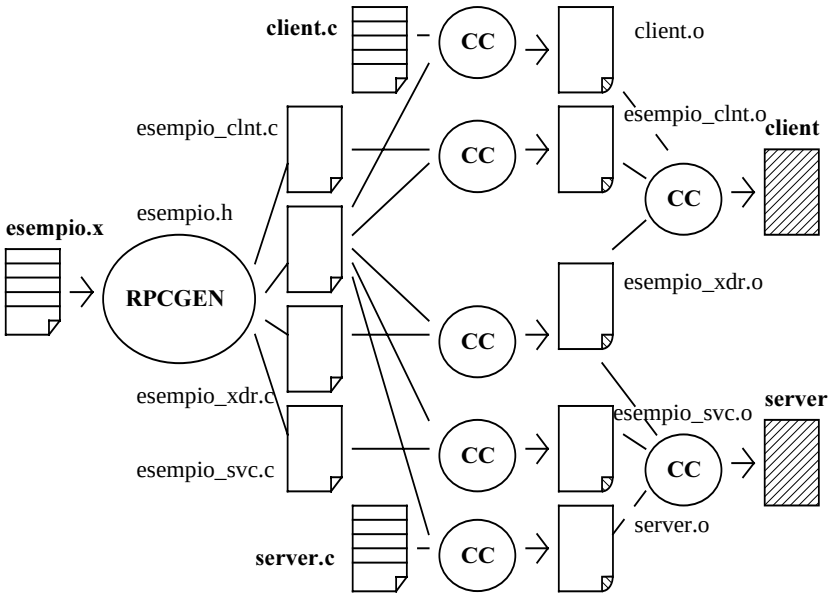
l'interfaccia RPC di comunicazione

data una specifica di partenza

file di linguaggio RPC esempio.**x**

si producono

- file di testata (header) esempio**.h**
- file stub del client esempio**_clnt.h**
- file stub del server esempio**_svc.h**
- file di routines XDR esempio**_xdr.h**



Normali fasi di sviluppo e compilazione per l'applicazione distribuita

Esempio di programma con una chiamata locale per stampare messaggi su video:

```
# include <stdio.h>

main(argc,argv)
     int argc;     char *argv[];
{    char *message;
if (argc !=2) {fprintf(stderr,"uso:%s <messaggio>\n", argv[0]);
                exit(1);     }
message = argv[1];
if (! printmessage (message))
{/* chiamata locale al servizio di stampa su video. */
     fprintf(stderr,"%s: errore sulla stampa.\n", argv[0]);
     exit(1); }
printf("Messaggio consegnato.\n");
exit(0);
}

printmessage (msg)
/* procedura locale per il servizio di stampa su video. */
     char *msg;
{    FILE *f;
f = fopen("/dev/console","w");
if (f == NULL) return(0);
fprintf(f,"%s\n",msg);  fclose(f);  return(1);
}
```

In caso remoto si deve trasformare

Definizione del programma RPC

RPCGEN usa un file con estensione .x

Due parti descrittive in linguaggio RPC:

1. ***definizioni di programmi RPC***: specifiche del protocollo RPC per i servizi offerti, cioè l'identificazione dei servizi ed il tipo dei parametri
2. ***definizioni XDR***: definizioni dei tipi di dati dei parametri. Presenti solo se il tipo di dato non ha una corrispondente funzione built-in

file msg.x

```
program MESSAGEPROG {
   version MESSAGEVERS {
      int PRINTMESSAGE(string) = 1;
      } = 1;
   } = 0x20000013;
```

Una procedura PRINTMESSAGE, versione 1
argomento di tipo string e risultato di tipo intero

Per le specifiche del protocollo RPC:

- il numero di procedura **zero** (0) è riservato dal protocollo RPC per la NULLPROC
- ogni definizione di procedura ha un solo **parametro d'ingresso e d'uscita**
  Il passaggio del parametro in ingresso è *per valore* e il risultato *per riferimento*
- gli identificatori di programma, versione e procedura usano lettere maiuscole

Sviluppo del servizio remoto

Si deve produrre un file con le definizioni necessarie agli stub per il linguaggio C

Qui
  solo definizioni di **programma**
  tipi di dati con funzioni ***built-in*** di conversione XDR

/* ***msg.h*** */

```
#define MESSAGEPROG ((u_long)0x20000013)
#define MESSAGEVERS ((u_long)1)
#define PRINTMESSAGE ((string)1)

extern printmessage_1();
```

Il file viene incluso dai due stub generati client e server

In caso di nuovi tipi di dati si devono definire le nuove **strutture dati** per le quali si devono generare le nuove **funzioni di trasformazione**

Sviluppo della procedura di servizio

Il codice del servizio è identico alla versione locale

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "msg.h"    /*  file prodotto da RPCGEN con msg.x */

/* procedura che implementa il servizio. */
int * printmessage_1 (msg)
char **msg;
/* argomento è passato per indirizzo. */
{    static int result;
     FILE *f;
  f=open("/dev/console","w");
  if (f==NULL) {   result=0;    return(&result);}
  fprintf(f,"%s\n",*msg);  fclose(f);  result=1;
  return(&result);
/* restituzione del risultato per indirizzo. */
}
```

Differenze

- si accetta un **unico puntatore** come argomento e restituisce un **puntatore unico** come risultato
- il risultato punta ad una *variabile statica*, per permanere oltre la chiamata della procedura
- *il nome della procedura cambia*
  si aggiunge il carattere underscore seguito dal numero di versione (in caratteri minuscoli)

Sviluppo del programma principale cliente

Il cliente invoca i servizi

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "msg.h"
main(argc,argv)          /* programma principale client */
int argc;  char *argv[];

{    CLIENT *cl;         int *result;
     char *server;       char *message;

if (argc<3) {     fprintf (stderr,"uso: %s host msg\n", argv[0]);
                  exit(1); }
server = argv[1];  message = argv[2];
cl = clnt_create (server, MESSAGEPROG, MESSAGEVERS,
                  "tcp");
if (cl==NULL)
{    clnt_pcreateerror (server); exit(1); }
result = printmessage_1 (&message,cl);
/* invocazione nello stub del cliente*/
if (result==NULL) {     clnt_perror (cl,server); exit(1);}

if (* result == 0)
    { fprintf(stderr,"%s: %s problema\n", argv[0],  server);
              exit(1);    }
printf("Messaggio consegnato a %s\n",server);
}
```

Si noti la creazione di un **gestore di trasporto client** per il protocollo TCP

il protocollo di trasporto potrebbe essere anche assunto a default

Variazioni rispetto al caso locale

Creazione di un gestore di trasporto per il client:

CLIENT * **clnt_create** (host, prog, vers, protocol)
char *host; u_long prog,vers; char *protocol;

host == nome del nodo remoto
prog == programma remoto
vers == versione
protocol == protocollo (*"udp"* o *"tcp"*)

Come in ***clntudp/tcp_create***()

- il nome della procedura è identificato come per il server
  il nome della procedura cambia
  si aggiunge il carattere underscore seguito dal numero di versione (in caratteri minuscoli)

- gli argomenti della procedura server sono due:
  uno è quello *vero e proprio*
  l'altro è il *gestore client*

Linguaggio RPC

Linguaggio dichiarativo di specifica dei dati e della interazione per RPC
due sottoinsiemi di definizioni

1. **definizioni di tipi di dati**
definizioni XDR per generare le definizioni in C e le relative funzioni per la conversione in XDR

2. definizioni delle **specifiche di protocollo RPC**
definizioni di programmi RPC per il protocollo RPC (identificazione del servizio e parametri di chiamata)

Esempio:

```
/* definizioni XDR */
const MAXNAMELEN=256;
const MAXSTRLEN=255;
struct r_arg {
  string filename<MAXNAMELEN>;
  int start;  int length;};
struct w_arg {
  string filename <MAXNAMELEN>;
  opaque block<>;  int start;};
struct r_res {  int errno;  int reads;
                opaque block<>;};
struct w_res {  int errno;  int writes;};
/* definizione di programma RPC */
program ESEMPIO {
   version ESEMPIOV {
   int   PING(int)=1;
   r_res READ(r_arg)=2;
   w_res WRITE(w_arg)=3;
   }=1;
}=0x20000020;
```

Definizioni XDR

Solamente tipi
linguaggio di descrizione da tradurre in C

Dichiarazioni di tipi atomici del linguaggio C con aggiunte

- ***bool***

due valori: TRUE e FALSE
==> tradotto nel tipo *bool_t*

- ***string*** con due utilizzi
  - con specifica del *numero massimo* di caratteri <fra angle-brackets>
  - lunghezza *arbitraria* con <angle-brackets vuoti>

string nome<30>; *tradotto in* `char *nome;`
string cognome<>; *tradotto in* `char *cognome;`

Diverse funzioni XDR generate dal compilatore per gestire la conversione (xdr_string())

- ***opaque*** una sequenza di bytes senza un tipo di appartenenza (con o senza la massima lunghezza)

opaque buffer<512>; *tradotto da RPCGEN in*

```
struct {    u_int buffer_len;
            char *buffer_val;
          } buffer;
```

opaque file<>; *tradotto da RPCGEN in*

```
struct {    u_int file_len;
            char *file_val;
          } file;
```

Differenza nelle funzioni XDR generate (xdr_bytes())

- ***void*** : Non si associa il nome della variabile di seguito

Dichiarazioni di tipi semplici

Analoga alla dichiarazione in linguaggio C

```
simple-declaration:
   type-ident variable-ident
```

Identificatore *type-ident*
o ***un tipo atomico*** *o* ***un tipo in linguaggio RPC***

NON si esegue un controllo di tipo
Se il tipo indicato non appartiene a uno dei due insiemi, il compilatore RPCGEN assume che sia definito a parte
==>
le funzioni di conversione XDR sono assunte **esterne**

colortype color; *tradotto da RPCGEN in*

```
colortype color;
```

Dichiarazione di vettori a lunghezza fissa

```
fixed-array-declaration:
   type-ident variable-ident "[" value "]"
```

colortype palette[8]; *tradotto da RPCGEN in*

```
colortype palette[8];
```

Dichiarazione vettori a lunghezza variabile

```
variable-array-declaration:
   type-ident variable-ident "<" value ">"
   type-ident variable-ident "<" ">"
```

Si può
- specificare la *lunghezza massima* del vettore
- lasciare la lunghezza *arbitraria*

int heights <12>; *tradotto da RPCGEN in*

```
struct {
        u_int heights_len;
        int *heights_val;
    } widths;
```

int widths <>; *tradotto da RPCGEN in*

```
struct {
        u_int widths_len;
        int *widths_val;
    } widths;
```

Struttura con due campi
con suffisso **_len e _val**
il primo contiene il numero di posizioni occupate
il secondo è un puntatore ad un vettore con i dati

Dichiarazione di tipi puntatori

```
pointer-declaration:
   type-ident "*" variable-ident
```

Supporto offerto al trasferimento di strutture recursive
listitem *next; *tradotto da RPCGEN in*

```
listitem *next;
```

RPCGEN fornisce il supporto per il trasferimento
- *stessa dichiarazione di variabile di tipo puntatore*
- **funzione XDR** dedicata a ricostruire il riferimento indicato dal puntatore una volta trasferito il dato sull'altro nodo

Definizione di tipi struttura

```
struct-definition:
   "struct" struct-ident "{"
      declaration-list
   "}"

declaration-list:
   declaration ";"
   declaration ";" declaration-list
```

Struttura XDR	Struttura C
struct coordinate { **int x;** **int y;** **};**	**struct coordinate {** **int x;** **int y;** **};** **typedef struct coordinate coordinate;**

Definizione di tipi unione

```
union-definition:
   "union" union-ident "switch"
   "(" simple-declaration ")"
      "{"   case-list "}"

case-list:
   "case" value ":" declaration ";"
   "default"    ":" declaration ";"
   "case" value ":" declaration ";" case-
list
```

Unione XDR	Unione C

union read_result switch (int errno)
{
case 0: opaque data[1024];
default: void;
};
	struct read_result {
	int errno;
	union { char data[1024];
	} read_result_u;
	};
	typedef struct read_result read_result;

Definizione di tipi enumerazione

```
enum-definition:
   "enum" enum-ident "{"
      enum-value-list "}"

enum-value-list:
   enum-value
   enum-value "," enum-value-list

enum-value:
   enum-value-ident
   enum-value-ident "=" value
```


Enumerazione

Enumerazione XDR	Enumerazione C

enum colortype {
RED = 0,
GREEN = 1,
BLUE = 2
};
	enum colortype {
	RED = 0,
	GREEN = 1,
	BLUE = 2
	};
	typedef enum colortype colortype;

Definizione di tipi costante

Costanti simboliche

```
const-definition:
   "const" const-ident "=" integer
```

Ad esempio, nella specifica di dimensione di un vettore

Costante XDR	Costante macro-processore C
const MAXLEN = 12;	#define MAXLEN 12

Definizione di tipi non standard

```
typedef-definition:
   "typedef" declaration
```

Definizione XDR di tipo	Definizione C di tipo
typedef string fname_type <255>;	typedef char *fname_type;

Definizione di Programmi RPC

specifiche di protocollo RPC

- identificatore unico del servizio offerto
- modalità d'accesso alla procedura mediante i parametri di chiamata e di risposta

```
program-definition:
   "program" program-ident
      "{"   version-list"}" "=" value

version-list:
   version ";"
   version ";" version-list

version:
   "version" version-ident
   "{"  procedure-list "}" "=" value

procedure-list:
   procedure ";"
   procedure ";" procedure-list

procedure:
   type-ident procedure-ident
      "(" type-ident ")" "=" value
```

il compilatore, genera le due procedure stub

Definizione di programma RPC	Definizione di protocollo RPC in C

program TIMEPROG {
  version TIMEVERS {
  usigned int TIMEGET(void) = 1;
  void TIMESET(unsigned int) = 2;
  } = 1;
} = 44;
	#define TIMEPROG ((u_long) 44)
	#define TIMEVERS ((u_long) 1)
	#define TIMEGET ((u_long) 1)
	#define TIMESET ((u_long) 2)

Esempio

Servizio di *remote directory list* (RLS)
una procedura remota che fornisce la lista dei files di un direttorio di un file system su nodo remoto
strutture recursive come le liste

Definizione del programma RPC

/* file rls.x */

```
const MAXNAMELEN = 255;
typedef string nametype <MAXNAMELEN>;
/* argomento della chiamata. */
typedef struct namenode *namelist;
struct namenode {      nametype name;  namelist next; };
/* risultato del servizio RLS. */
union readdir_res switch (int errno) {
    case 0:       namelist list;
    default:      void;
    };

program RLSPROG {
    version RLSVERS {
        readdir_res READDIR (nametype)=1;
        } = 1;
    } = 0x20000013;
```

Definizione di struttura recursiva basata su puntatori XDR
Considerare la gestione dei segnali nel server
SIGHUP, SIGINT, SIGQUIT, SIGTERM

File prodotto da RPCGEN

/* rls.h */

```
#define MAXNAMELEN 255
typedef char *nametype;
bool_t xdr_nametype ();
typedef struct namenode * namelist;
bool_t  xdr_namelist ();
struct namenode {      nametype name;  namelist next;};
typedef struct namenode namenode;
bool_t xdr_namenode ();
struct readdir_res {    int errno;
                        union { namelist list;} readdir_res_u;
                    };
typedef struct readdir_res readdir_res;
bool_t xdr_readdir_res ();

#define RLSPROG ((u_long)0x20000013)
#define RLSVERS ((u_long)1)
#define READDIR ((u_long)1)
extern readdir_res *readdir_1 ();
```

Il file è incluso dai due stub generati client e server

/* *rls_xdr.c*: routine di conversione XDR */

```
#include <rpc/rpc.h>
#include "rls.h"
bool_t  xdr_nametype (xdrs, objp)
    XDR *xdrs;     nametype *objp;
{ if (!xdr_string (xdrs, objp, MAXNAMELEN))
        {return (FALSE);} return (TRUE);
}

/* in questa funzione vengono impiegati i puntatori XDR */
bool_t xdr_namelist (xdrs, objp)
    XDR *xdrs;     namelist *objp;
{ if (! xdr_pointer (xdrs, (char **)objp,
            sizeof(struct namenode), xdr_namenode))
        {return (FALSE);} return (TRUE);
}

bool_t  xdr_namenode (xdrs, objp)
    XDR *xdrs;     namenode *objp;
{   if (!xdr_nametype(xdrs, &objp->name))
            {return (FALSE);}
    if (!xdr_namelist(xdrs, &objp->next)) {return (FALSE);}
    return (TRUE);
}

bool_t  xdr_readdir_res (xdrs, objp)
    XDR *xdrs;     readdir_res *objp;
{   if (!xdr_int(xdrs, &objp->errno)) {return (FALSE);}
    switch (objp->errno)  {
    case 0: if (!xdr_namelist(xdrs, &objp->readdir_res_u.list))
                    {return (FALSE);}
            break;          }
    return (TRUE);
}
```

/* *rls_clnt.c*: stub del cliente */

```
#include <rpc/rpc.h>
#include <sys/time.h>
#include "rls.h"
/* direttive varie per il compilatore C */
#ifdef hpux
     memset(&res, 0, sizeof(res));
#else  hpux
     memset(&res, sizeof(res));
#endif hpux
#ifdef hpux

/* assegnamento  time-out per la chiamata */
static struct timeval TIMEOUT = { 25, 0 };

readdir_res * readdir_1 (argp, clnt)
     nametype *argp;
     CLIENT *clnt;
{    static readdir_res res;
if ( clnt_call (clnt, READDIR, xdr_nametype, argp,
                    xdr_readdir_res, &res, TIMEOUT)
                         != RPC_SUCCESS)
               {return (NULL);}
     return (&res);
}
```

Reale chiamata remota nello stub

/* rls_svc.c: stub del server */

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

static void rlsprog_1 ();  /funzione di dispatching */

main()
{    SVCXPRT * transp;
pmap_unset (RLSPROG, RLSVERS);
/* creazione di un gestore di trasporto UDP  */
transp = svcudp_create (RPC_ANYSOCK);
if (transp == NULL) {
          fprintf(stderr, "cannot create udp service.\n");
          exit(1); }
/* registrazione con il protocollo UDP */
if (! svc_register (transp, RLSPROG, RLSVERS,  rlsprog_1,
                              IPPROTO_UDP))
       {  fprintf(stderr, "unable to register (RLS P, V, udp).\n");
          exit(1); }

/* creazione di un gestore di trasporto TCP */
transp = svctcp_create (RPC_ANYSOCK, 0, 0);
if (transp == NULL) {
          fprintf(stderr, "cannot create tcp service.\n");
          exit(1); }
/* registrazione con il protocollo TCP  */
if (! svc_register (transp, RLSPROG, RLSVERS, rlsprog_1,
                              IPPROTO_TCP))
       {  fprintf(stderr, "unable to register (RLSP, V, tcp).\n");
          exit(1); }
svc_run();
fprintf(stderr, "svc_run returned\n");     exit(1);
}
```

```
/* procedura di dispatching */
static void rlsprog_1  (rqstp, transp)
     struct svc_req *rqstp;    SVCXPRT *transp;
{    union { nametype readdir_1_arg; } argument;
     char *result;
     bool_t (*xdr_argument)(), (*xdr_result)();
     char *(*local)();
/*  sono diventati generici: local, procedura da invocare,
   argument e result i parametri di ingresso e uscita, le funzioni
   xdr xdr_argument e xdr_result */
switch (rqstp->rq_proc) {
     case NULLPROC:
          svc_sendreply (transp, xdr_void, NULL);
          return;
     case READDIR:
          xdr_argument = xdr_nametype;
          xdr_result = xdr_readdir_res ;
          local = (char *(*)()) readdir_1;
          break;
     default:
          svcerr_noproc (transp);
          return;
                              }
memset(&argument, 0, sizeof(argument)); /* in caso HP-UX */
if (!svc_getargs  (transp, xdr_argument, &argument))
     {svcerr_decode (transp); return; }
result = (*local) (&argument, rqstp);
if ((result != NULL) &&
               !svc_sendreply(transp, xdr_result, result))
               {svcerr_systemerr (transp);}
if (!svc_freeargs (transp, xdr_argument, &argument))
     { fprintf(stderr, "unable to free arguments\n");exit(1); }
}
```


Indirettezza della chiamata al servizio locale

si vogliono generare stub dedicati a più servizi con diversi tipi di argomenti e risultati

Argomento ==> **union**
la stessa area di memoria ad ognuno dei possibili tipi di dati, cioè gli argomenti dei servizi

La variabile ***argument*** dal messaggio di protocollo RPC avviene per indirizzo
Se la variabile ***argument*** fosse passata per valore alla procedura locale, la procedura locale dovrebbe mantenere una union

Variabile ***result*** ==> puntatore a carattere

stub *rls_clnt.c* offre al client la procedura *readdir_1()*

stub *rls_svc.c* contiene la registrazione dei servizi in RPC, sia come servizi TCP che come servizi UDP
e la procedura di dispatching *rlsprog_1()* che chiama il servizio vero e proprio *readdir_1()*

Sviluppo del programma principale cliente

/* file rls.c */

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

extern int errno;
CLIENT * clnt_create ();

main(argc,argv)
     int argc;      char *argv[];
{    CLIENT *cl;    namelist nl;
     char *server;    char *dir;   readdir_res *result;

if (argc!=3) {fprintf(stderr,"uso: %s <host> <dir>\n",argv[0]);
                exit(1);     }
server = argv[1];  dir = argv[2];

cl = clnt_create (server, RLSPROG, RLSVERS, "tcp");
if (cl==NULL) { clnt_pcreateerror (server); exit(1); }

result= readdir_1 (&dir,cl);
if (result==NULL) { clnt_perror (cl,server);exit(1); }

if (result->errno!=0) {  errno=result->errno;
                         perror (dir);  exit(1);}

/* stampa risultati */
for (nl=result -> readdir_res_u. list; nl != NULL; nl= nl -> next
)
     {printf("%s\n",nl->name);}
}
```


Sviluppo del server

/* file rlsserver.c */

```
#include <rpc/rpc.h>
#include <sys/dir.h>
#include <stdio.h>
#include "rls.h"
extern int errno;
extern char *malloc();
extern char *strcpy();

readdir_res * readdir_1 (dirname)
     nametype *dirname;
{    DIR *dirp;  struct direct *d;  namelist nl;
     namelist *nlp;   static readdir_res res;
/* la chiamata di questa funzione libera la struttura per
   cancellare gli effetti della chiamata precedente */
xdr_free (xdr_readdir_res,&res);

dirp = opendir (*dirname);/* apertura di una directory */
if (dirp==NULL) { res.errno=errno; return(&res); }
nlp=&res.readdir_res_u.list;
/* inizia la creazione di una struttura recursiva */
while (d= readdir (dirp))
{         nl=*nlp = (namenode*) malloc(sizeof(namenode));
          nl->name = malloc(strlen(d->d_name)+1);
          strcpy(nl->name,d->d_name); nlp = &nl->next;    }
* nlp=NULL;
/* chiusura della lista con un puntatore a NULL */
res.errno=0;  closedir (dirp);    /* chiusura della directory */
return(&res);
}
```

STATICO vs. DINAMICO

Si noti il taglio **statico** degli strumenti

Al momento dello sviluppo si devono conoscere tutti i tipi e le funzioni XDR corrispondenti

E in caso **DINAMICO**?
Molte informazioni possono essere passate attraverso comunicazioni

- numero di programma
  comunicato nel primo messaggio

- tipi utilizzati nelle procedure
  i tipi e le funzioni XDR dovrebbero essere integrabili dinamicamente

Necessità di una entità capace di fornire:
- realizzazione delle operazioni
- specifiche di trasformazione dei tipi
- aggancio dinamico ai nomi dei servizi

Strumento di **name service**

vedi **CORBA**