



**Università degli Studi di Bologna
Facoltà di Ingegneria**

**Corso di
Reti di Calcolatori L-A**

Implementazione RPC

Luca Foschini

Anno accademico 2009/2010

RPC: motivazioni e modelli

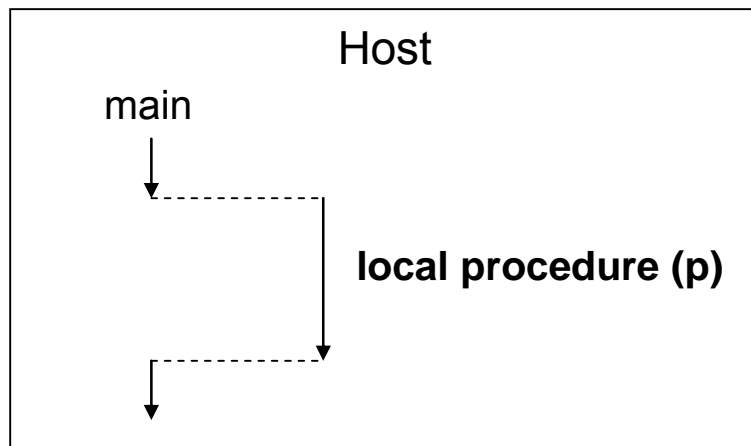
Possibilità di invocare una procedura non locale
operazione che interessa un nodo remoto e ne richiede un servizio

Modello di riferimento → CLIENTE/SERVITORE
invocazione di un **servizio remoto**
con **parametri tipati, valore di ritorno**

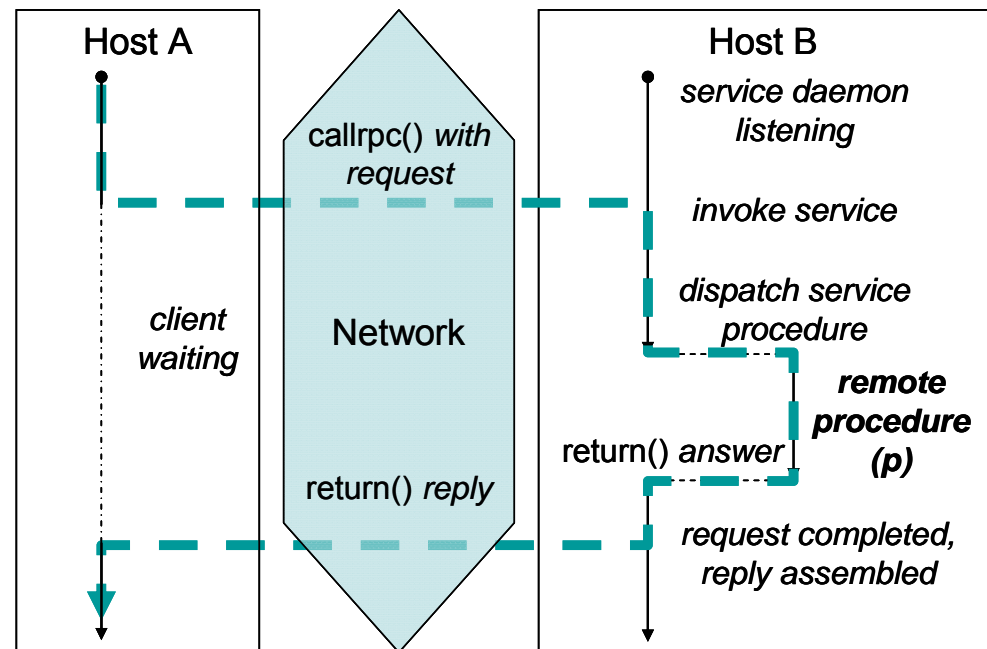
Modello di base: P.B. Hansen (Distributed Processing)
Nelson Birrel

Funzionamento invocazione remota

Invocazione **locale**
della procedura p



Invocazione **remota**
della procedura p



Requisiti per implementazione RPC

Il supporto **scambia messaggi** per consentire
identificazione dei messaggi di chiamata e risposta
identificazione unica della procedura remota

Il supporto **gestisce l'eterogeneità dei dati** scambiati
marshalling/unmarshalling argomenti;
serializzazione argomenti;

Il supporto **gestisce** alcuni **errori** dovuti alla distribuzione
implementazione errata
errori dell'utente
roll-over (ritorno indietro)

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)

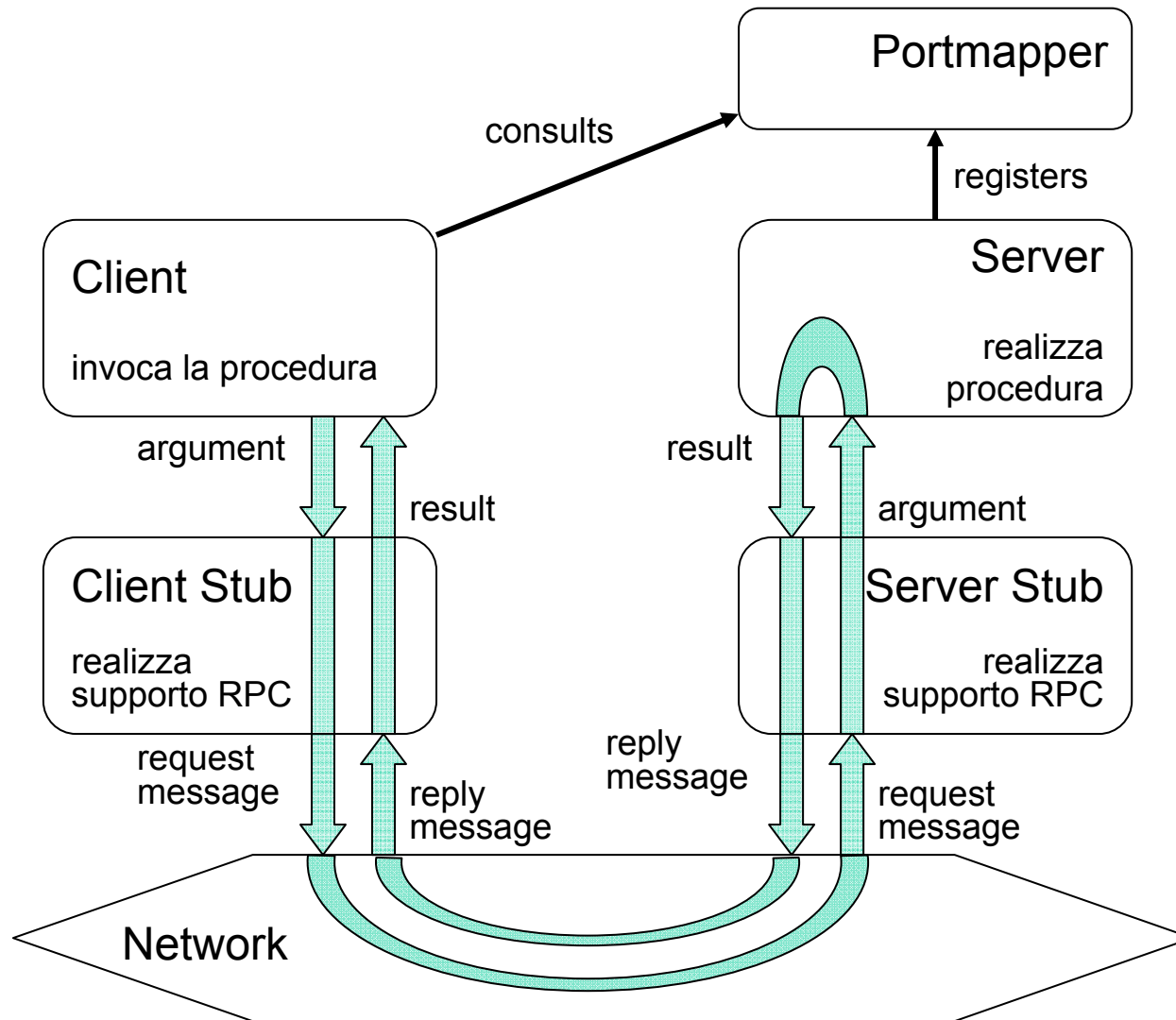
paralleli

operazioni per il **CLIENTE**

sincrone (SUN RPC)

asincrone

Architettura



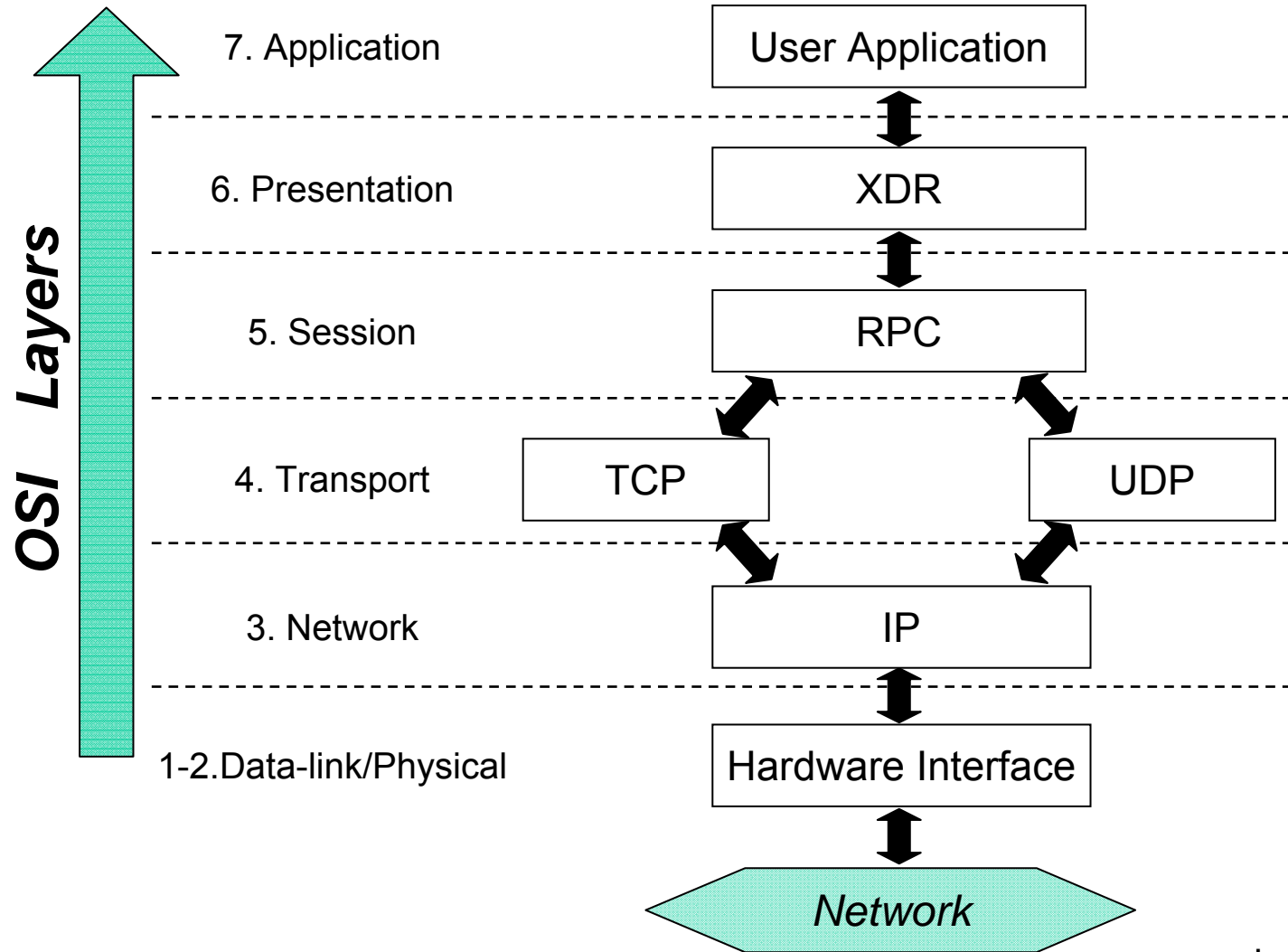
Implementazione del supporto

Noi vediamo l'implementazione e il supporto RPC di **Sun Microsystems: Open Network Computing (ONC)**

ONC è una suite di prodotti che include:

- **eXternal Data Representation (XDR)** →
rappresentazione e conversione dati
- **Remote Procedure Call GENerator (RPCGEN)** →
generazione del client e server stub
- **Portmapper** → risoluzione indirizzo del servitore
- **Network File System (NFS)** →
file system distribuito di Sun

SUN RPC e stack OSI



Definizione del programma RPC

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 è un tipo noto in RPC (per i dettagli vedere più avanti...)

Raggruppate in un unico file con estensione .x

Ad esempio per il programma **esempio** tutte le definizioni contenute in **esempio.x**

Definizione del servizio remoto

Diamo tutte le definizioni nel file `stampa.x`

```
/* stampa.x */  
program STAMPAPROG {  
    version STAMPAVERS {  
        int PRINTMESSAGE(string) = 1;  
    } = 1;  
} = 0x20000013;
```

Tralasciando per ora **program** e **version** (che vediamo dopo), questo pezzo di codice **definisce la procedura PRINTMESSAGE**, con **argomento di tipo string** e **risultato di tipo int (intero)**

Si noti che:

- ogni definizione di procedura ha **un solo** parametro d'ingresso e **un solo** parametro d'uscita
- gli **identificatori** (nomi) usano **lettere maiuscole**
- ogni procedura è associata ad un **numero di procedura unico** all'interno di un programma

Implementazione del programma RPC

Il programmatore deve sviluppare:

1. **il programma client**: implementazione del `main()` e della logica necessaria per **reperimento** e **binding** del servizio/i remoto/i → file `esempio.c`
2. **il programma server**: implementazione di tutte le **procedure** (servizi)
→ file `esempio_svc_proc.c`

NOTA: lo sviluppatore **NON** realizza il `main()`...

Chi invoca la procedura remota (lato server)?

RPC: un primo esempio

Servizio di stampa di messaggi su video locale In locale

```
# 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 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 come segue...**

Sviluppo della procedura di servizio

Il **codice** del servizio è (quasi) **identico** alla versione locale

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "msg.h" /* prodotto da RPCGEN */
int * printmessage_1_svc (msg, rp)
char **msg; /* argomento è passato per indirizzo */
struct svc_req * rp;
{ 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 risultato per indirizzo */
}
```

Differenze

- sia l'argomento di **ingresso** che l'**uscita** vengono passati **per riferimento**
- il **risultato** punta ad una **variabile statica** (allocazione **globale**, si veda oltre), per essere presente anche oltre la chiamata della procedura
- il **nome** della procedura **cambia leggermente** (si aggiunge il carattere underscore seguito dal numero di versione, tutto in caratteri minuscoli)

Sviluppo del programma principale: CLIENTE

Il cliente viene invocato col nome dell'**host remoto** e il **messaggio da stampare** a video e richiede il servizio di **stampa a video remota**

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "msg.h" /* prodotto da RPCGEN */
main(argc,argv) /* programma 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, STAMPAPROG, STAMPAVERS, "udp");
  if (cl==NULL) { clnt_pcreateerror (server); exit(1); }
  result = printmessage_1(&message,cl); /* invocazione client stub */
  /* Controllo esito richiesta, dettagli più avanti */
  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);
}
```

Variazioni rispetto al caso locale

Si noti la creazione del **gestore di trasporto client** (si veda oltre per ulteriori dettagli): il gestore di trasporto (`cl`) gestisce la comunicazione col server, in questo caso utilizzando un trasporto senza connessione (“`udp`”)

Il client **deve conoscere**:

- l'**host remoto** dove è in esecuzione il servizio;
- alcune informazioni per invocare il servizio stesso (**programma, versione e nome della procedura**)

Per l'**invocazione della procedura remota**:

- **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: i) **l'argomento di ingresso** vero e proprio e ii) il **gestore di trasporto del client**

Il client **gestisce gli errori** che si possono presentare durante l'invocazione remota usando le funzionalità di stampa degli errori: `clnt_pcreateerror` e `clnt_perror`

Passi di sviluppo di SUN RPC

1. **Definire servizi e tipi di dati** (se necessario) → `esempio.x`
2. **Generare** in modo automatico gli **stub** del **client** e del **server** e (se necessario) le **funzioni di conversione XDR** → uso di **RPCGEN**
3. **Realizzare** i programmi **client** e **server** (`client.c` e `esempio_svc_proc.c`), compilare tutti i file sorgente (programmi client e server, stub, e file per la conversione dei dati), e fare il **linking** dei file oggetto
4. **Pubblicare i servizi** (lato server)
 1. attivare il **Portmapper**
 2. registrare i servizi presso il Portmapper (attivando il server)
5. **Reperire** (lato client) **l'endpoint del server** tramite il Portmapper e **creare il gestore di trasporto** per l'interazione col server

A questo punto **l'interazione fra client e server può procedere**

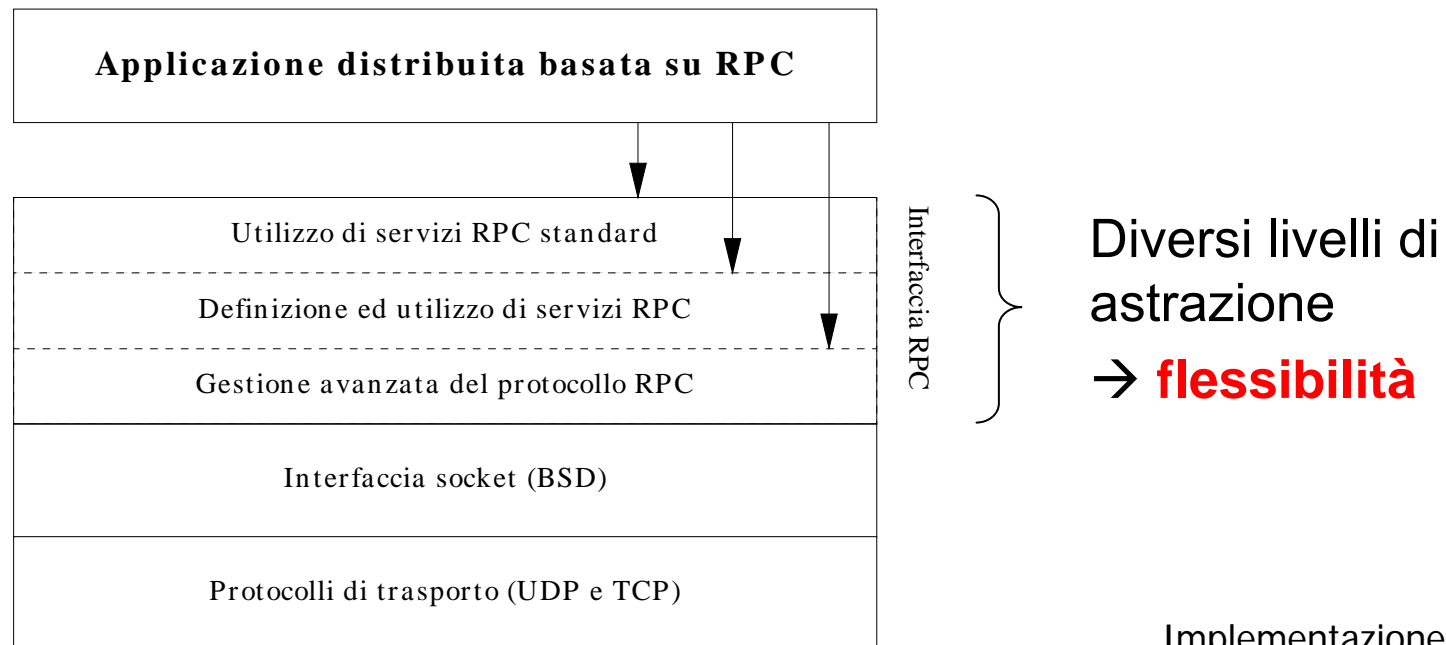
N.B.: questa è una **descrizione di base**, **dettagli** su RPCGEN, Portmapper e gestore di trasporto **saranno dati nel seguito**

Prima di vedere più in dettaglio il funzionamento del **supporto RPC** proviamo a fare un **confronto con Java RMI...**

RPC e sistemi 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

In caso di **RPC SUN**

- Un **programma** tipicamente contiene **più procedure remote**
- Sono previste anche **versioni multiple** delle procedure
- Un **unico argomento in ingresso** ed **in uscita** per ogni invocazione

Semantica e controllo concorrenza

Mutua esclusione garantita dal programma (e server): **non** si prevede alcuna **concorrenza** a default nell'ambito dello stesso programma server

→ **Server sequenziale** ed **una sola invocazione eseguita per volta**

Fino al ritorno al programma cliente, il **processo cliente** è **in modo sincrono bloccante** in attesa della risposta

Rischi e problemi RPC

Considerando servitori sequenziali

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

Naturalmente, più chiamate RPC possono manifestarsi in parallelo su un nodo servitore... che le serve una alla volta

e **SERVIZI PARALLELI??**

Potrebbero consentire risparmio e throughput maggiore
si possono realizzare? E come?

Semantica e affidabilità

Uso di protocollo UDP (**default SUN**)

Semantica at-least-once: a default, si fanno un certo numero di ritrasmissioni dopo un **intervallo di time-out** (ad esempio, 4 secondi)

Identificazione di RPC

Messaggio RPC deve contenere

numero di programma

numero di versione

numero di procedura

per la identificazione globale

Numeri di programma (32 bit)

- **0 - 1ffffffh** predefinito Sun → applicazioni d'interesse comune
- **20000000h - 3ffffffh** definibile dall'utente → applicazioni debug dei nuovi servizi
- **40000000h - 5ffffffh** 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

Livello alto RPC

Utilizzo di servizi RPC standard

Esempi di procedure pronte per essere invocate da un nodo ad un altro

Routine RPC	Descrizione
<i>rnusers()</i>	Fornisce il numero di utenti di un nodo
<i>rusers()</i>	Fornisce informazioni sugli utenti di un nodo
<i>rstat()</i>	Ottiene dati sulle prestazioni verso un nodo
<i>rwall()</i>	Invia al nodo un messaggio
<i>getmaster()</i>	Ottiene il nome del nodo master per NIS
<i>getrpcport()</i>	Ottiene informazioni riguardo agli indirizzi TCP legati ai servizi RPC

Numeri di programma noti

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

ESEMPIO: invocazione di rnuser()

```
/* Nota: utilizzare la libreria corretta in fase di linking, ad
esempio rpcsvc */
#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) //unico parametro -> hostname remoto
    {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 nuovi servizi RPC

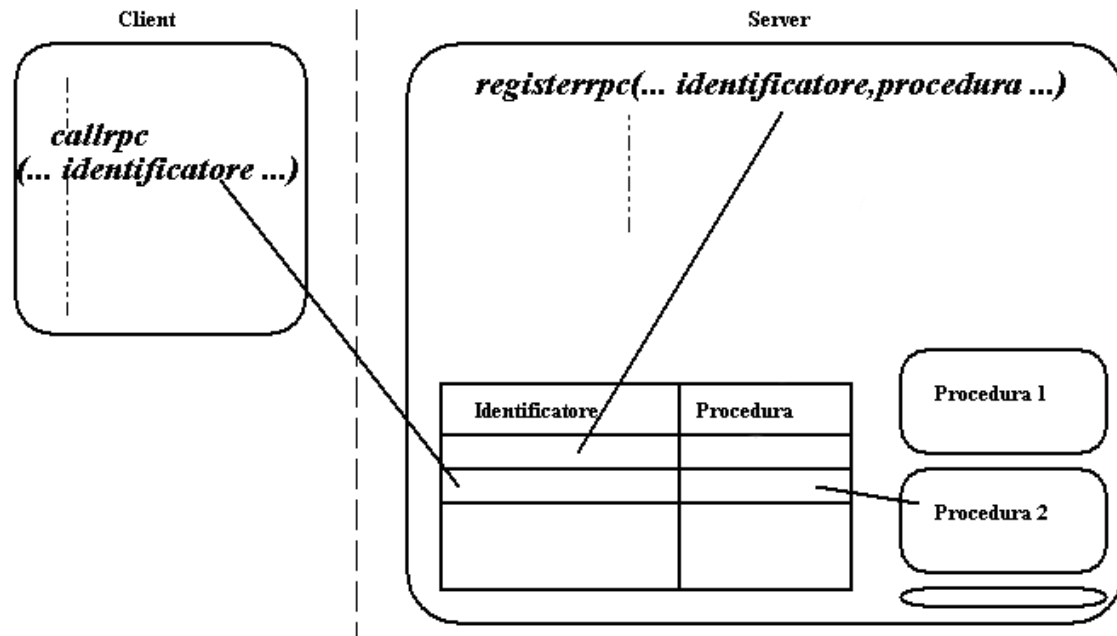
Due primitive: `callrpc()` e `registerrpc()`

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 compiute per ogni invocazione di procedura dalla parte del servitore

1. **ricerca** in tabella della procedura
2. recapito richiesta al programma ritrovato



CLIENT: richiesta esecuzione RPC

Primitiva callrpc: due parametri → un argomento e un risultato

```
int callrpc (  remotehost, n_prog, n_vers, n_proc,
              xdr_arg, arg, xdr_res, res )
char        *remotehost;  → nome del nodo remoto
u_long      n_prog,n_vers,n_proc;  → identificazione procedura
xdrproc_t   xdr_arg; → tipo argomento di ingresso
char        *arg; → argomento di ingresso (passaggio per valore)
xdrproc_t   xdr_res; → tipo risultato (per la funzione XDR)
char        *res; → risultato
```

Problemi nel passaggio di strutture a lista: il ricevente deve ricostruire la struttura dinamica (vedi dettagli uso **XDR** più avanti)

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: esempi

```
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_VERSIONMISMATCH=6,        /* rpc versions not compatible */
RPC_AUTHERROR=7,              /* authentication error */
RPC_PROGUNAVAIL=8,             /* program not available */
RPC_PROGVERSIONMISMATCH=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,           /* unknown protocol */
/* create errors */
RPC_PMAPFAILURE=14,            /* pmapper failed in its call */
RPC_PROGNOTREGISTERED=15,     /* remote program not registered */
...};
```

Inoltre: possibilità di
**terminazione per
timeout**



Semantica at-least-once
(default con UDP): prima
di dichiarare time-out, il
supporto RPC esegue
alcune ritrasmissioni in
modo trasparente

Livello intermedio: esempio invocazione servizio RPC standard – lato client

Invocazione di **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);
}
```

Livello intermedio: realizzazione servizio remoto utente – lato client

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) {... 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); }
```

Semplice operazione aritmetica sull'argomento (incremento di “100”): **intero in ingresso e risultato intero in uscita**

Livello intermedio: realizzazione servizio remoto utente – lato server

Implementazione procedura remota: le funzioni del protocollo RPC ricevono l'argomento e lo mettono in un'area di memoria del server passando l'indirizzo al server

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

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

Inoltre... è necessario **registrare la procedura remota** (dettagli dopo...)

```
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);
}
```

Omogeneità dei dati

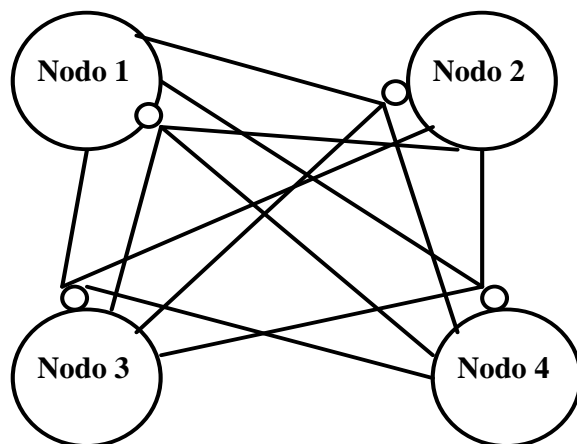
Per comunicare tra **nodì eterogenei** due soluzioni

1. Dotare ogni nodo di **tutte le funzioni di conversione** possibili per ogni rappresentazione dei dati

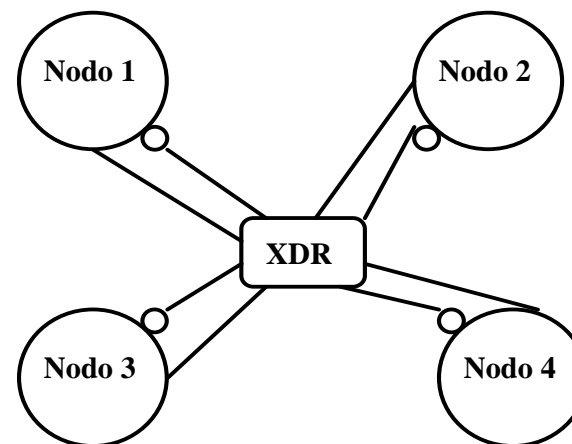
- **alta performance** e **alto numero** di funzioni di conversione $\rightarrow N*(N-1)$

2. Concordare un **formato comune** di rappresentazione dei dati: ogni nodo possiede le funzioni di conversione da/per questo formato

- **minore performance**, ma **basso numero** di funzioni di conversione $\rightarrow 2*N$



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)

eXternal Data Representation (XDR)

Funzioni svolte da protocollo XDR

→ **Marshalling**

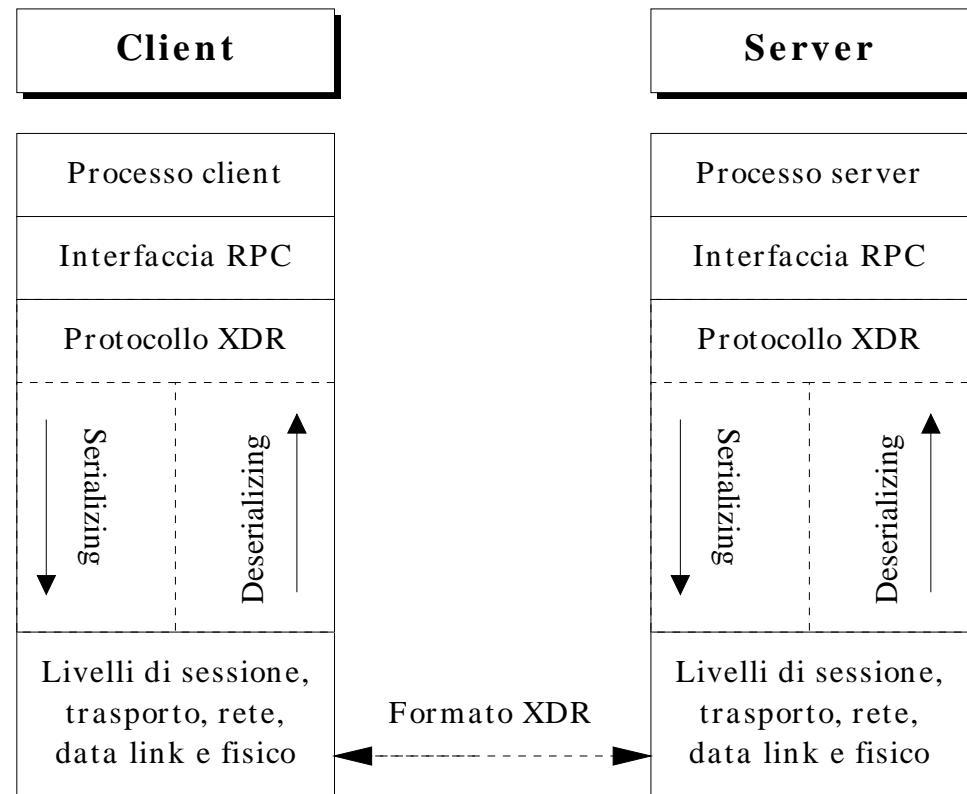
Collocazione in OSI

→ **Presentazione**

XDR procedure *built-in* di conversione, relative a:

- Tipi atomici predefiniti
- Tipi standard (costruttori riconosciuti)

NOTA: la primitiva ***callrpc*** accetta solo un argomento ed un solo risultato → necessità di definire una struttura che li raggruppa



Funzioni di conversione

Funzioni built-in

Funzione built-in	Tipo di dato
<i>xdr_bool()</i>	Logico
<i>xdr_char()</i> <i>xdr_u_char()</i>	Carattere
<i>xdr_short()</i> <i>xdr_u_short()</i>	Intero a 16 bit
<i>xdr_enum()</i>	Enumerazione
<i>xdr_float()</i>	Virgola mobile
<i>xdr_int()</i> , <i>xdr_u_int</i>	Intero
<i>xdr_long()</i> , <i>xdr_u_long()</i>	Intero a 32 bit
<i>xdr_void()</i>	Nulla
<i>xdr_opaque()</i>	Opaco (raw byte)
<i>xdr_double()</i>	Doppia precisione

Funzioni per tipi composti

Funzione	Tipo di dato
<i>xdr_array()</i>	Vettori con elementi di tipo qualsiasi
<i>xdr_vector()</i>	Vettori a lunghezza fissa
<i>xdr_string()</i>	Sequenza di caratteri con terminatore a NULL <i>N.B.: stringa in C</i>
<i>xdr_bytes()</i>	Vettore di bytes senza terminatore
<i>xdr_reference()</i>	Riferimento ad un dato
<i>xdr_pointer()</i>	Riferimento ad un dato, incluso NULL
<i>xdr_union()</i>	Unioni

Funzioni di utilità XDR: esempio

Funzioni XDR per la conversione di numeri interi, vedi file `rpc/xdr.h`

```
bool_t  xdr_int (xdrs, ip)
    XDR *xdrs; int *ip;
```

Le funzioni xdr restituiscono **valore vero** se la conversione **ha successo**

Anche con tipi definiti dall'utente, ad esempio **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 a sua volta xdr.h */
```

```
bool_t  xdr_simple (xdrsp, simplep)
```

```
XDR *xdrsp;  /* rappresenta il tipo in formato XDR */
```

```
struct simple *simplep;  /* rappresenta il formato interno */
```

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

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

```
    return(TRUE);
```

```
}
```

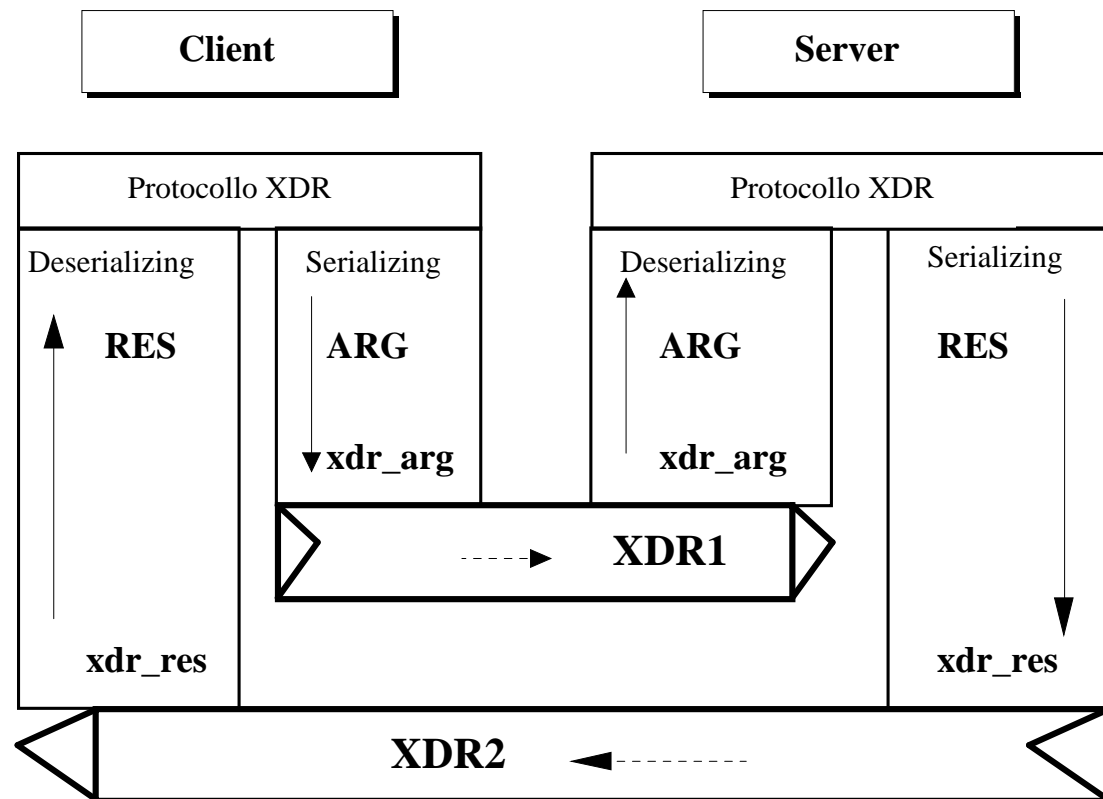

Stream XDR

Per ogni informazione da trasmettere si hanno due trasformazioni → sulla rete il **solo formato 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 sono usate anche per la **sola trasformazione dei dati** (e non associate alla comunicazione)



Flussi XDR

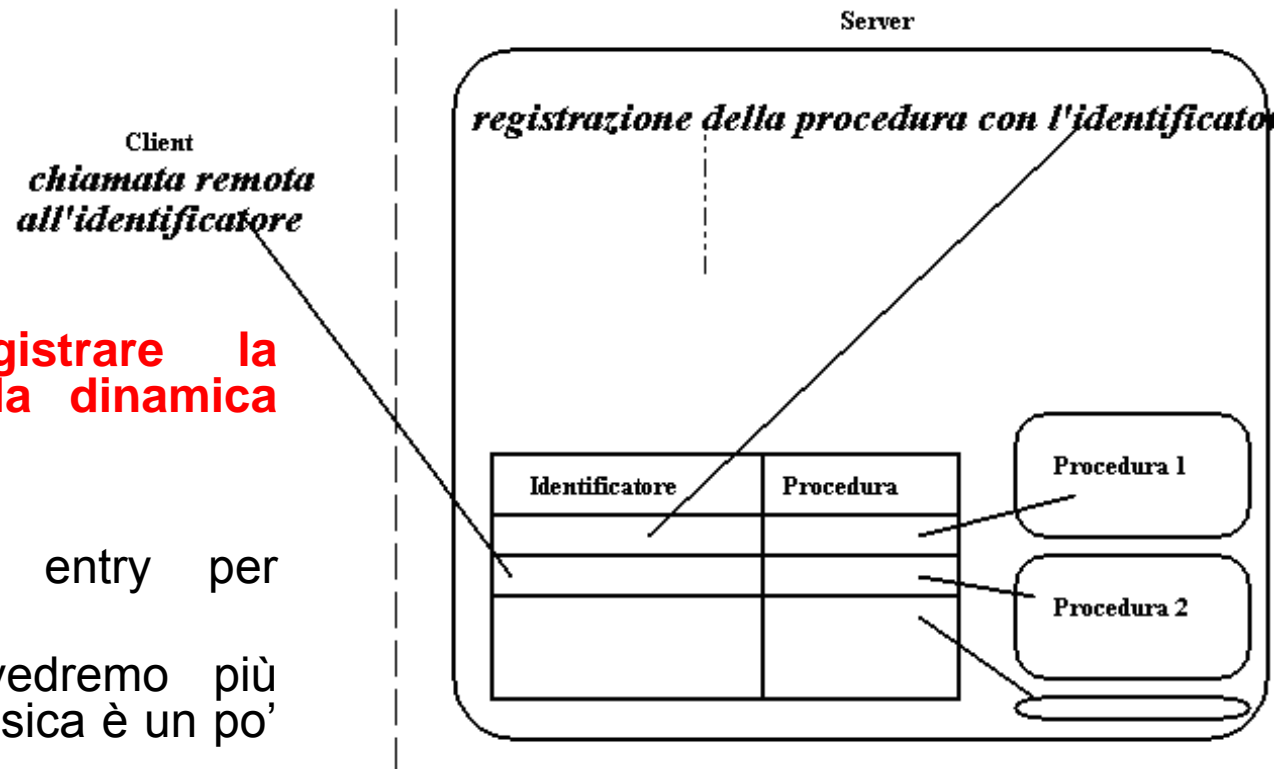
Server: registrazione procedura remota

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

Concettualmente una entry per procedura...

... **in realtà** (come vedremo più avanti), la registrazione fisica è un po' diversa



Primitiva di registrazione: **registerrpc**

```
int registerrpc (n_prog, n_vers, n_proc, proc_name, xdr_arg, xdr_res)
u_long n_prog,  n_vers,  n_proc;
char *(* proc_name ) ();
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**

Primitiva svc_run

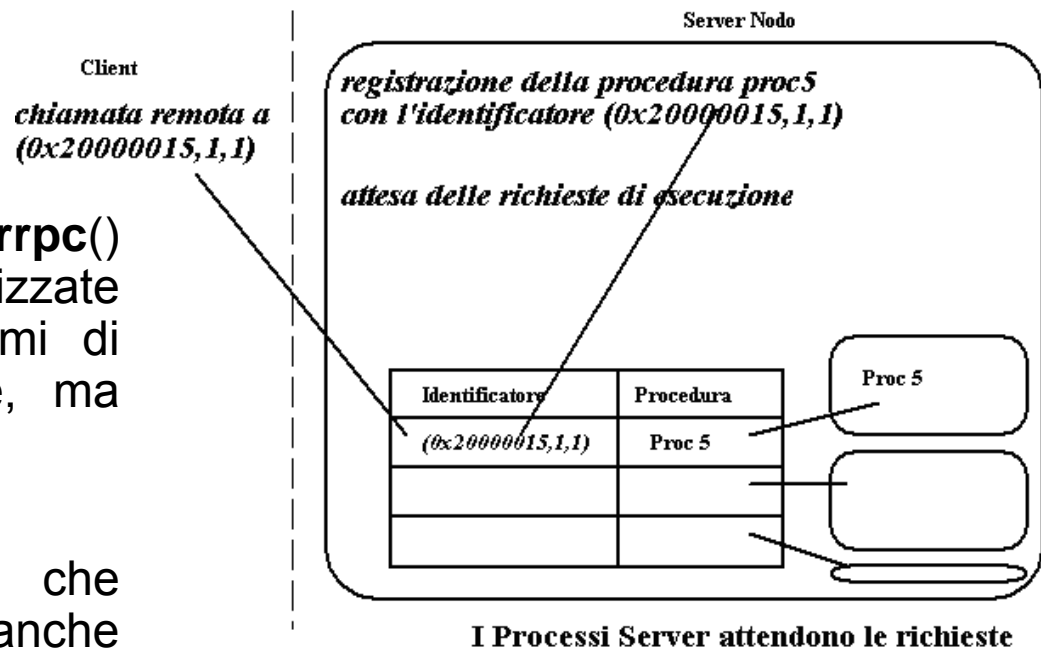
Un server diventa un demone mettendosi in attesa di tutte le possibili richieste
⇒ **svc_run()**

Dopo la registrazione della procedura, il processo che ha eseguito la primitiva **registerrpc()** deve rimanere in attesa di chiamate: la **svc_run** che esprime la attesa del server

La primitiva **svc_run()** non termina

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

Uso di **processi servitori** che contengono i servizi (procedure), 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**

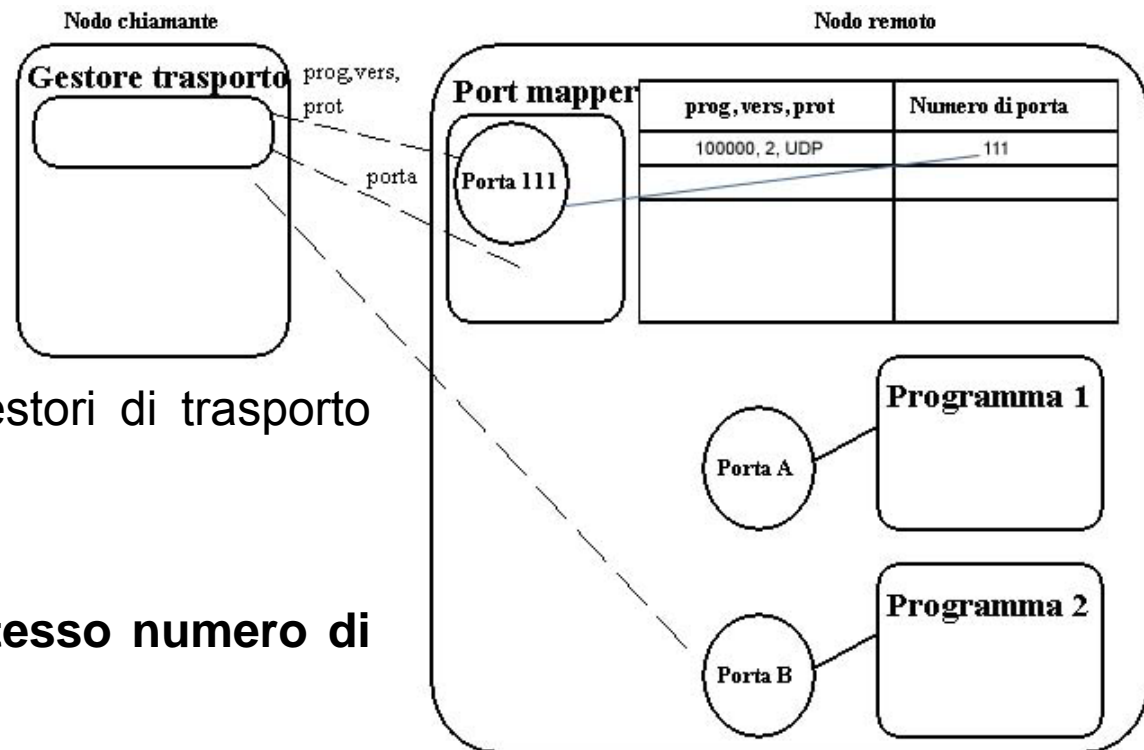
Manca il numero di procedura dato che tutti i servizi RPC (procedure all'interno di un programma) condividono lo stesso **gestore di trasporto**

La tabella è strutturata come insieme di elementi:

```
struct mapping {  
    unsigned long prog;  
    unsigned long vers;  
    unsigned int prot; // uso costanti IPPROTO_UDP ed IPPROTO_TCP  
    unsigned int port; / corrispondenza con la porta assegnata  
};  
/* implementazione a lista dinamica della tabella */  
struct *pmaplist { mapping map; pmaplist next;}
```

PORT MAPPER

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

PORT MAPPER: motivazioni e dettagli

Per limitare il numero di porte riservate → un solo processo sulla porta 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 le seguenti procedure:

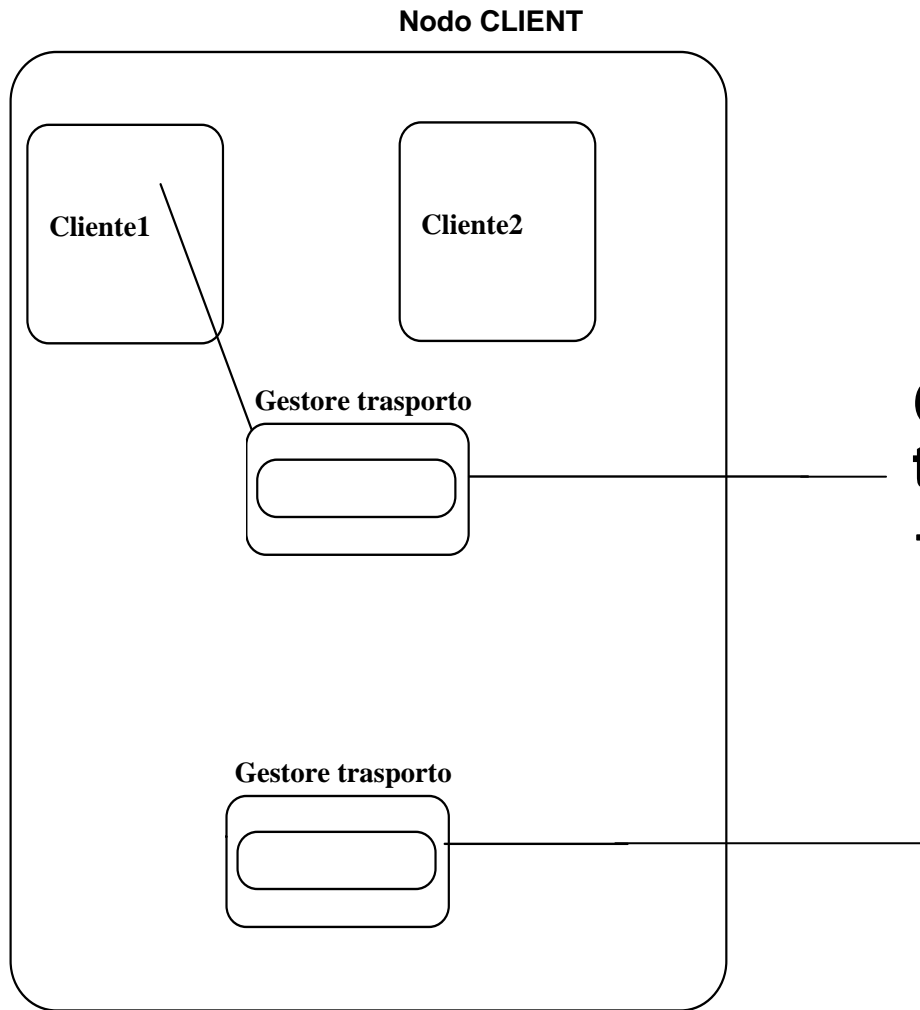
- **Inserimento** di un servizio
- **Eliminazione** di un servizio
- **Corrispondenza associazione astratta e porta**
- **Intera lista** di corrispondenza
- **Supporto** all'esecuzione remota

A default utilizza solo il trasporto **UDP**

Limiti: dal cliente **ogni chiamata interroga il port mapper** (poi attua la richiesta)

Per avere la lista di tutti i servizi invocabili su di un host remoto si provi ad invocare il comando: `rpcinfo -p nomehost`

Architettura e funzioni RPC: lato client



Creazione di un gestore di trasporto

→ Per mantenere il collegamento RPC con i clienti

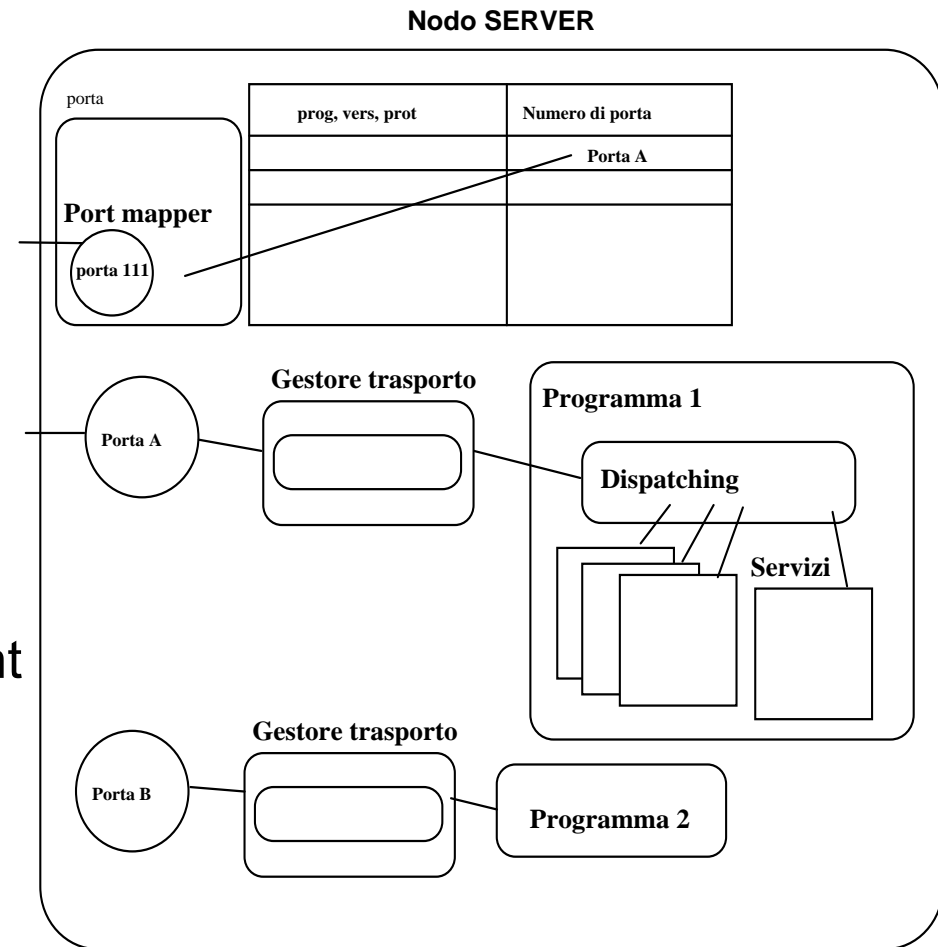
Architettura e funzioni RPC: lato server

Creazione di un gestore di trasporto

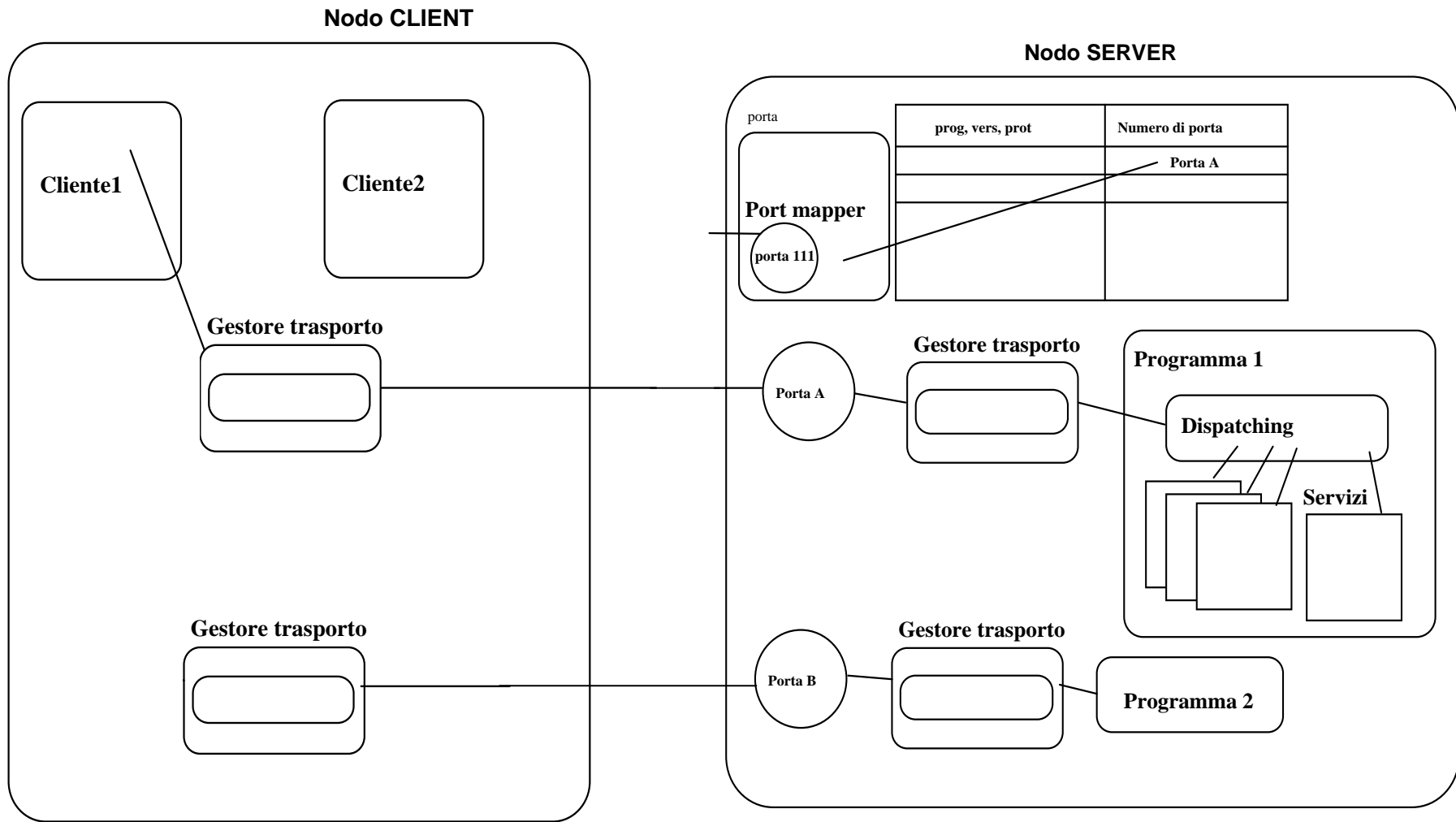
- Per mantenere il collegamento RPC con i clienti

Dispatching (locale) del messaggio RPC

- Per inoltrare la richiesta del client alla procedura corretta:
il **messaggio RPC** contiene il **numero della procedura**



Architettura e funzioni RPC



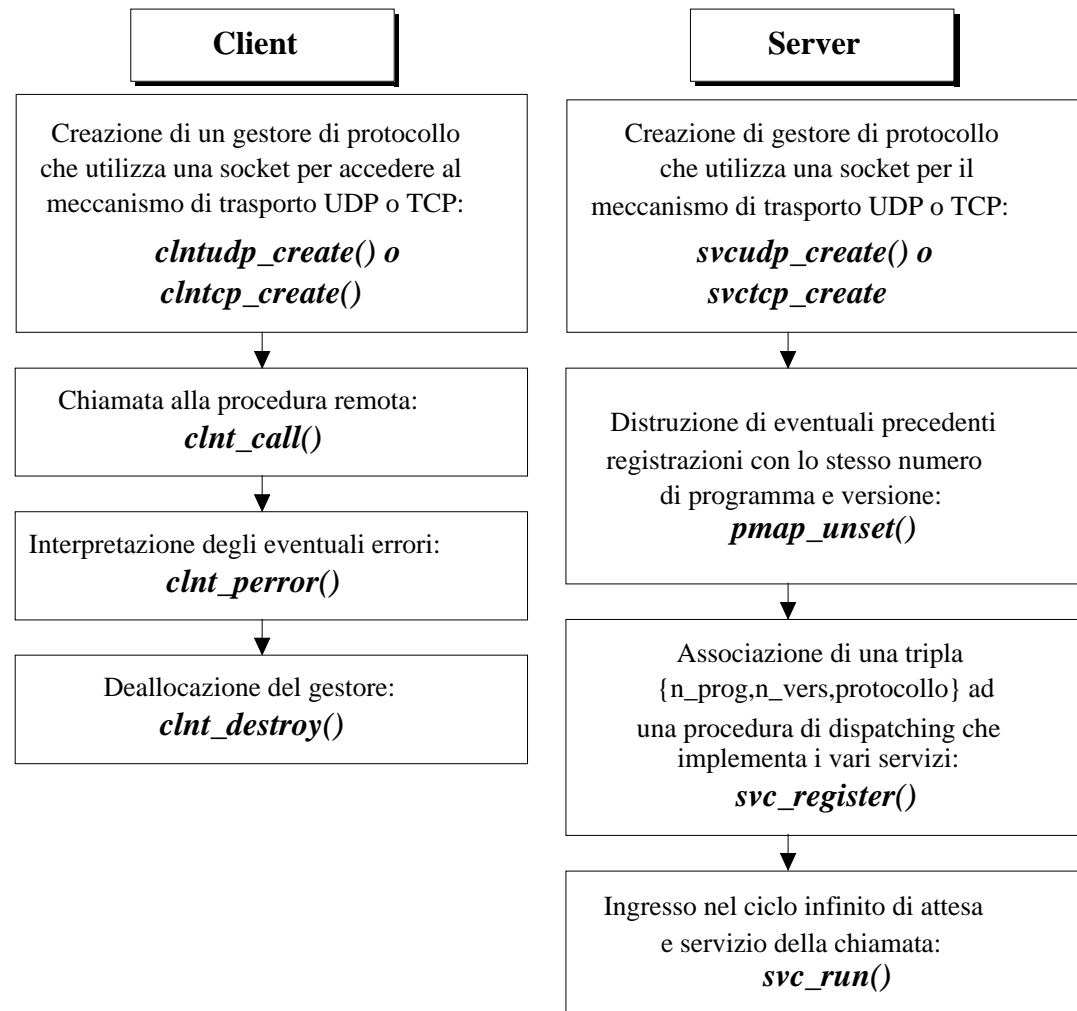
Livello basso RPC

Gestione avanzata del protocollo RPC

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

- `svcudp_create()` per **ottenere un gestore UDP** (default per SUN)
- `svctcp_create()` per **ottenere un gestore TCP** (in caso di protocollo affidabile)

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 assoc.*/
    struct xp_ops {
        bool_t (*xp_recv)(); /* ricezione richieste */
        enum xpstat (*xp_stat)(); /* stato del trasporto */
        bool_t (*xp_getargs)(); /* legge gli argomenti */
        bool_t (*xp_reply)(); /* invia una risposta */
        bool_t (*xp_freeargs)(); /* libera memoria allocata */
        void (*xp_destroy)(); /* distrugge la struttura */
    } * xp_ops;
    int xp_addrlen; /* lunghezza ind. 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;
```

SERVER: gestore di trasporto (ancora)

È una **struttura astratta** che

- **contiene puntatori 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;
```

svcudp_create(): se **sock** vale → **RPC_ANYSOCK** si crea un datagram socket per i risultati

Altrimenti, il parametro è un socket descriptor (collegato 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

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 (**svc_req**)

```
struct svc_req {  
    u_long    rq_prog;           /* numero di programma */  
    u_long    rq_vers;          /* versione */  
    u_long    rq_proc;          /* procedura richiesta */  
    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 la procedura da svolgere

rq_xprt identifica la struttura dati del gestore di trasporto, dalla quale è possibile (si veda struttura **SVCXPTR**):

- ricavare i parametri per l'esecuzione tramite **svc_getargs()**
- spedire la risposta tramite la **svc_sendreply()**

SERVER: procedura di dispatching (ancora)

```
bool_t svc_getargs (xprt,inproc,in)
    SVCXPRT *xprt;
    xdrproc_t inproc;    /* routine XDR */
    char *in;    /* puntatore a struttura dati */
```

```
bool_t svc_sendreply (xprt,outproc,out)
    SVCXPRT *xprt;
    xdrproc_t outproc;    /* routine XDR */
    char *out;    /* puntatore a struttura dati */
```

Funzioni XDR per conversioni formato argomenti e risultati

I valori di input (getargs) e di output (sendreply) sono 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: procedura di dispatching (ancora)

Associa numero programma e versione

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;
```

xprt	⇒	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: creazione di un gestore di trasporto (ancora)

```
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 e definisce dimensione 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 e 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;
```

clnt	⇒	gestore di trasporto locale
procnum	⇒	identificazione della procedura
inproc	⇒	tipo di argomenti
outproc	⇒	tipo di risultato
in	⇒	argomento unico
out	⇒	risultato unico
tout	⇒	tempo di attesa della risposta

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

CLIENT: analisi errori e rilascio risorse

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

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

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;
```

Ancora dettagli su XDR

Generalità XDR

Dichiarazioni di tipi atomici del linguaggio C con aggiunte

- **bool** con due valori: TRUE e FALSE → tradotto nel tipo *bool_t* con
- **string** con due utilizzi
 - con specifica del *numero massimo di caratteri* <fra angle-brackets>
 - lunghezza *arbitraria* con <angle-brackets vuoti>

Ad esempio: `typedef string nome<30>;` tradotto in `char *nome;`
 `typedef string nome<>;` tradotto in `char *nome;`

Diverse funzioni XDR generate dal compilatore per gestire la conversione (`xdr_string()`) → **Provare!!**

Oppure string utilizzato direttamente come tipo di in/output → in questo caso **viene generato un file xdr?**

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

`typedef opaque buffer<512>;` tradotto da *RPCGEN* in
`struct { u_int buffer_len; char *buffer_val; } buffer;`
`typedef 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

ATTENZIONE!! RPCGEN NON 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**

Ad esempio: `typedef colortype color; // tradotto da RPCGEN in colortype color;`

Dichiarazione vettori a lunghezza fissa

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

Ad esempio:

`typedef colortype palette[8]; // tradotto da RPCGEN in colortype palette[8];`

Dichiarazione di matrici

In XDR **non** è possibile definire direttamente strutture innestate, **ma** bisogna sempre passare per definizioni di strutture intermedie

Ad esempio, la definizione della seguente struttura dati (matrice) **non viene interpretata correttamente** da rpcgen che ritorna errori e non riesce a terminare:

```
struct MatriceCaratteri{ char matrice [10][20]; };
```

Mentre passando da una **struttura dati intermedia**, si come quella qui sotto, **rpcgen termina correttamente**:

```
struct RigaMatrice{ char riga [20]; };
```

```
struct MatriceCaratteri{ RigaMatrice riga [10]; };
```

Si facciano alcune prove scrivendo i file .x e generando i file per le trasformazioni dati e i file .h con rpcgen ...

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, oppure
- lasciare la **lunghezza arbitraria**

Ad esempio:

```
typedef int heights <12>;
```

tradotto da RPCGEN in: struct { u_int heights_len; int *heights_val; } heights;

```
typedef int widths <>;
```

tradotto da RPCGEN in: struct { u_int widths_len; int *widths_val; } widths;

Stessa 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**

Cambia la **funzione xdr di conversione**

Dichiarazione di tipi puntatori

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

Supporto offerto al trasferimento di strutture ricorsive

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;

Dichiarazione di tipi unione

union-definition:

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

Unione XDR	Unione C
<pre>union read_result switch (int errno) { case 0: opaque data[1024]; default: void; };</pre>	<pre>struct read_result { int errno; union { char data[1024]; } read_result_u; }; typedef struct read_result read_result;</pre>

case-list:

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

case-list

Definizione di tipi enumerazione

enum-definition:

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

Enumerazione XDR	Enumerazione C
<pre>enum colortype { RED = 0, GREEN = 1, BLUE = 2 };</pre>	<pre>enum colortype { RED = 0, GREEN = 1, BLUE = 2 }; typedef enum colortype colortype;</pre>

enum-value-list:

```
enum-value
enum-value "," enum-value-list
```

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

Dichiarazione di tipi costante

Costanti simboliche

Costante XDR	Costante macro-processore C
<code>const MAXLEN = 12;</code>	<code>#define MAXLEN 12</code>

`const-definition:`

`"const" const-ident "=" integer`

Ad esempio, nella **specifica di dimensione di un vettore**

Definizione di tipi non standard

`typedef-definition:`

`"typedef" declaration`

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

Definizione di Programmi RPC

Specifiche protocollo RPC

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

Definizione di programma RPC	Definizione di protocollo RPC in C
<pre>program TIMEPROG { version TIMEVERS { unsigned int TIMEGET(void) = 1; void TIMESET(unsigned int) = 2; } = 1; } = 44;</pre>	<pre>#define TIMEPROG ((u_long) 44) #define TIMEVERS ((u_long) 1) #define TIMEGET ((u_long) 1) #define TIMESET ((u_long) 2)</pre>

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**

Uso SUN RPC: Dettagli

Esempio completo di utilizzo di XDR

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:

```
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;};
program ESEMPIO {    /* definizione di programma RPC */
    version ESEMPIOV {
        int    PING(int)=1;
        r_res READ(r_arg)=2;
        w_res WRITE(w_arg)=3;
    }=1;
}=0x200000020;
```

Generazione Automatica delle RPC

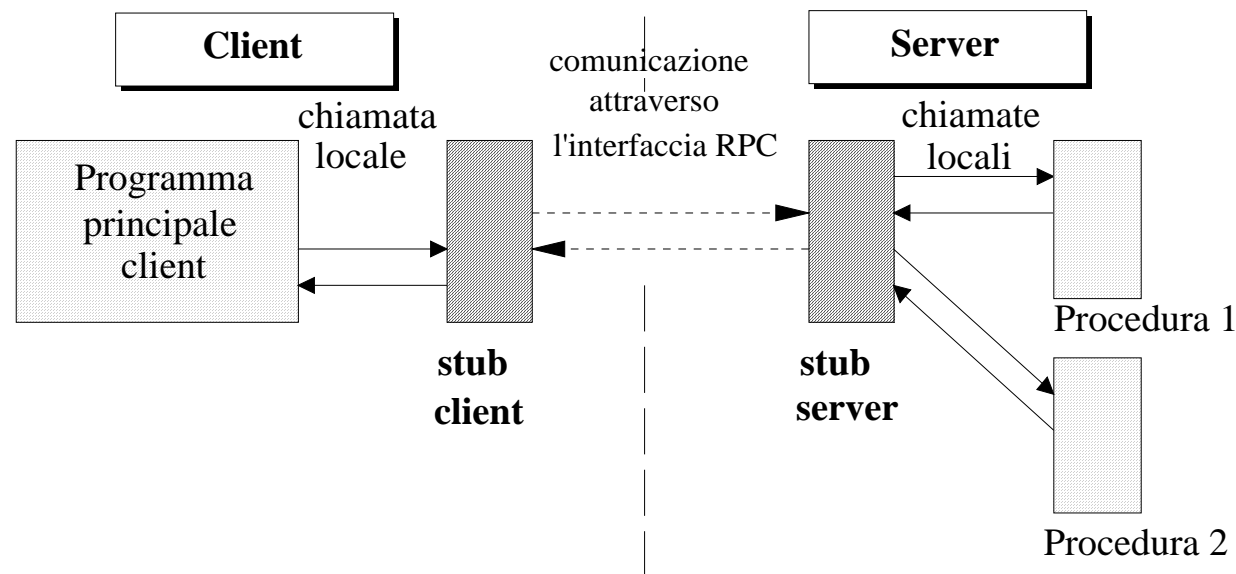
Maggiore astrazione dell'applicazione → uso di procedure **stub**

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 XDR**



 Parti sviluppate direttamente dal programmatore

 Parti fornite da RPCGEN

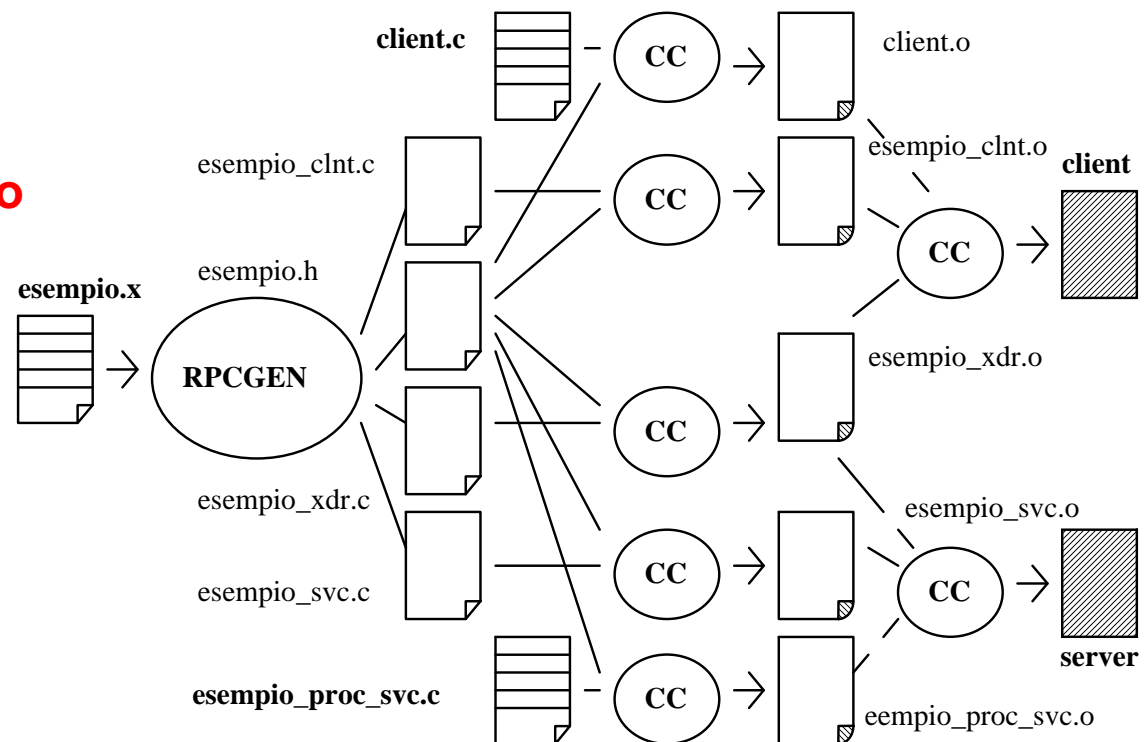
Processo di sviluppo

Data una **specifica di partenza**

- file di linguaggio XDR
→ esempio.x

RPCGEN produce **in automatico**

- file di testata (header)
→ esempio.h
- file stub del client
→ esempio_clnt.c
- file stub del server
→ esempio_svc.c
- file di routine XDR
→ esempio_xdr.c



Lo **sviluppatore deve realizzare**

- programma client → **client.c**
- programma server → **esempio_svc_proc.c**

ESEMPIO COMPLETO: file .x

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 ricorsive come le liste

```
/* file rls.x: definizione del programma RPC */
const MAXNAMELEN = 255;
typedef string nametype <MAXNAMELEN>; /* argomento della chiamata. */

typedef struct namenode *namelist;
struct namenode { nametype name; namelist next; };

union readdir_res switch (int errno){ /*risultato del servizio RLS */
    case 0:          namelist list;
    default:         void;
};

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

Implementazione RPC 66

FILE .x OSSERVAZIONI

Prima parte del file → definizioni XDR

- delle **costanti**:
- dei **tipi di dati** dei parametri di ingresso e uscita per cui per tutti i tipi di dato per i quali non esiste una corrispondente funzione built-in

Seconda parte del file → definizioni XDR delle **procedure**

La procedura READDIR è la **procedura** numero 1 della **versione** 1 del **programma** numero 0x20000013 (espresso come numero esadecimale)

Si noti che 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**
- gli identificatori di **programma**, **versione** e **procedura** usano **tutte lettere maiuscole** e uno **spazio di nomi** che è il seguente:
 - **<NOMEPROGRAMMA>PROG** per il nome del programma
 - **<NOMEPROGRAMMA>VERS** per la versione del programma
 - **<NOMEPROCEDURA>** per i nomi delle procedure

File prodotti da RPCGEN: rls.h

Nel seguito vediamo più nel dettaglio il contenuto dei file prodotti automaticamente da RPCGEN

```
/* rls.h */
#include <rpc/rpc.h>
#ifdef __cplusplus
extern "C" {
#endif
#define MAXNAMELEN 255

typedef char *nametype;
typedef struct namenode *namelist;

struct namenode { nametype name; namelist next; };
typedef struct namenode namenode;

struct readdir_res {int remoteErrno; union {namelist list; }readdir_res_u; };
typedef struct readdir_res readdir_res;

#define RLSPROG 0x20000013
#define RLSVERS 1
#if defined(__STDC__) || defined(__cplusplus)
#define READDIR 1
```

File prodotti da RPCGEN: rls.h

```
extern readdir_res * readdir_1(nametype *, CLIENT *);
extern readdir_res * readdir_1_svc(nametype *, struct svc_req *);
extern int rlsprog_1_freeresult (SVCXPRT *, xdrproc_t, caddr_t);
```

```
#else /* K&R C */
#define READDIR 1
extern readdir_res * readdir_1();
extern readdir_res * readdir_1_svc();
extern int rlsprog_1_freeresult ();
#endif /* K&R C */
```

```
/* the xdr functions */
#if defined(__STDC__) || defined(__cplusplus)
extern bool_t xdr_nametype (XDR *, nametype*);
extern bool_t xdr_namelist (XDR *, namelist*);
extern bool_t xdr_namenode (XDR *, namenode*);
extern bool_t xdr_readdir_res (XDR *, readdir_res*);
```

```
#else /* K&R C */
extern bool_t xdr_nametype ();
extern bool_t xdr_namelist ();
extern bool_t xdr_namenode ();
extern bool_t xdr_readdir_res ();
#endif /* K&R C */
#ifdef __cplusplus
}
#endif
```

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 saranno generate in automatico le nuove **funzioni di trasformazione**

File prodotti da RPCGEN: rls_xdr.c

```
/* rls_xdr.c: routine di conversione XDR */
#include "rls.h"
bool_t  xdr_nametype (XDR *xdrs, nametype *objp)
{ register int32_t *buf;
  if (!xdr_string (xdrs, objp, MAXNAMELEN)) return FALSE;
  return TRUE; }

bool_t  xdr_namelist (XDR *xdrs, namelist *objp)
{ register int32_t *buf;
  if (!xdr_pointer (xdrs, (char **)objp, sizeof (struct namenode),
    (xdrproc_t) xdr_namenode)) return FALSE;
  return TRUE; }

bool_t  xdr_namenode (XDR *xdrs, namenode *objp)
{ register int32_t *buf;
  if (!xdr_nametype (xdrs, &objp->name)) return FALSE;
  if (!xdr_namelist (xdrs, &objp->next)) return FALSE;
  return TRUE; }

bool_t  xdr_readdir_res (XDR *xdrs, readdir_res *objp)
{ register int32_t *buf;
  if (!xdr_int (xdrs, &objp->remoteErrno)) return FALSE;
  switch (objp->remoteErrno) {
  case 0:
    if (!xdr_namelist (xdrs, &objp->readdir_res_u.list)) return FALSE; break;
  default:
    break; }
  return TRUE; }
```

File prodotti da RPCGEN: stub client

```
/* rls_clnt.c: stub del cliente */
#include <memory.h> /* for memset */
#include "rls.h"

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

readdir_res * readdir_1(nametype *argp, CLIENT *clnt)
{
    static readdir_res clnt_res;

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

NOTA: reale **chiamata remota nello stub**

File prodotti da RPCGEN: stub server

```
/* rls_svc.c: stub del server */
#include "rls.h"
#include <stdio.h>
...

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

int main (int argc, char **argv)
{
    register SVCXPRT *transp;
    /* de-registrazione di eventuale programma con stesso nome */
    pmap_unset (RLSPROG, RLSVERS);
    /* creazione gestore trasp. e registrazione servizio con UDP */
    transp = svcudp_create(RPC_ANYSOCK);
    if (transp == NULL)
    { fprintf(stderr, "%s", "cannot create udp service."); exit(1); }
    if (!svc_register(transp, RLSPROG, RLSVERS, rlsprog_1, IPPROTO_UDP))
    {fprintf (stderr, "%s", "unable to register..., udp)."); exit(1); }
    /* creazione gestore trasp. e registrazione servizio con TCP */
    transp = svctcp_create(RPC_ANYSOCK, 0, 0);
    if (transp == NULL){ fprintf(stderr, "%s", "..."); exit(1); }
    if (!svc_register(transp, RLSPROG, RLSVERS, rlsprog_1, IPPROTO_TCP))
    { fprintf (stderr, "%s", "unable to register ..., tcp)."); exit(1); }
    svc_run(); /* attivazione dispatcher */
    fprintf (stderr, "%s", "svc_run returned"); exit (1);
}
```

Implementazione RPC 72

File prodotti da RPCGEN: stub server

```
/* rls_svc.c (stub del server): procedura di dispatching */
static void rlsprog_1(struct svc_req *rqstp, register SVCXPRT *transp)
{
    union { nametype readdir_1_arg; } argument;
    char *result; xdrproc_t _xdr_argument, _xdr_result;
    char *(*local)(char *, struct svc_req *);

    /* 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:
        (void) svc_sendreply (transp, (xdrproc_t) xdr_void, (char *)NULL); return;
    case READDIR:
        _xdr_argument = (xdrproc_t) xdr_nametype;
        _xdr_result = (xdrproc_t) xdr_readdir_res;
        local = (char *(*)(char *, struct svc_req *)) readdir_1_svc; break;
    default:
        svcerr_noproc (transp); return;
    }
    memset ((char *)&argument, 0, sizeof (argument));
    if (!svc_getargs(transp, (xdrproc_t) _xdr_argument,
        (caddr_t) &argument)) { svcerr_decode (transp); return; }
    result = (*local)((char *)&argument, rqstp);
    if (result != NULL && !svc_sendreply(transp, (xdrproc_t) _xdr_result, result))
        {svcerr_systemerr (transp);}
    if (!svc_freeargs (transp, (xdrproc_t) _xdr_argument, (caddr_t) &argument))
        {... exit (1); } return;
}
```

Aspetti implementativi: passaggio parametri e invocazioni funzioni

Indirettezza della chiamata al servizio locale: si vogliono **generare stub dedicati** a più servizi con **diversi tipi di argomenti** e **risultati**

Argomento → **union**

Uso della stessa area di memoria **per tutti i possibili tipi di dati** (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 (procedura) vero e proprio **readdir_1_svc()**

File sviluppati dal programmatore: client

```
/* file client.c: il client*/
#include <stdio.h>
#include <rpc/rpc.h>
#include "rls.h"

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->remoteErrno!=0) {      perror (dir);  exit(1);}
    /* stampa risultati */
    for(nl=result->readdir_res_u.list; nl != NULL; nl = nl->next )
        printf("%s\n",nl->name);
}
```

OSSERVAZIONI

Creazione del 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: server
prog	⇒	programma remoto: RLSPROG
vers	⇒	versione: RLSVERS
protocol	⇒	protocollo: "tcp"

E' la chiamata di livello basso per la creazione del gestore di trasporto

Simile a `clntudp/tcp_create()`, ma lascia la definizione di alcuni parametri (es. tempo di wait) al supporto RPC

Invocazione della procedura

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

File sviluppati dal programmatore: server

```
/* file rls_svc_proc.c: il programma server */
#include <rpc/rpc.h>
#include <sys/dir.h>
...
extern int errno;

readdir_res * readdir_1_svc (dirname, rd)
    nametype * dirname; struct svc_req * rd;
{ DIR *dirp; struct direct *d; namelist nl;
  namelist *nlp; static readdir_res res;
  /* libero memoria allocata dalla chiamata precedente */
  xdr_free((xdrproc_t) xdr_readdir_res, (caddr_t) &res);
  dirp = opendir(*dirname); /* apertura di una directory */
  if( dirp==NULL ) { res.remoteErrno=errno; return &res; }
  nlp=&res.readdir_res_u.list;
  while (d= readdir (dirp)) /* creazione di una struttura recursiva */
  {   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.remoteErrno=0; closedir(dirp); /* chiusura directory */
  return &res;
}
```

A cosa serve **rd**?

Si riprenda la definizione della struttura **svc_req** vista sopra

Variabili static

Quali **effetti** produce la keyword **static** e perché è necessario dichiarare il **risultato** come variabile **static**?

Visibilità

- sono visibili dove sono state definite: **funzioni** o **moduli** (file)
- ma **non** sono visibili all'esterno

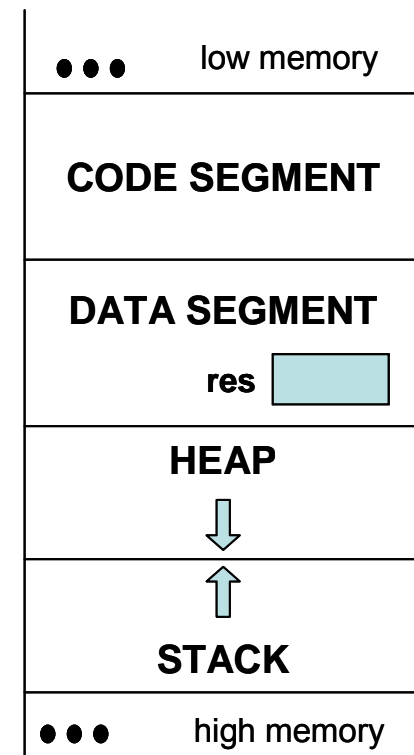
Tempo di vita

- allocazione **globale**
- tempo di vita **pari al programma**
 - Una variabile statica interna ad una funzione **permane oltre la singola invocazione** della procedura
 - Ogni invocazione della stessa procedura **utilizza il valore precedente** della variabile
- Politica di allocazione attraverso **dati statici**

Il valore di **ritorno deve essere static** in modo da essere disponibile e poter operare marshalling e spedizione del risultato al client quando la procedura termina

Altrimenti res terminata la procedura verrebbe **rimosso dallo stack**

E **quando allochiamo noi la struttura dati** (malloc)?



Bibliografia

J. Bloomer, “**Power Programming with RPC**”, Ed. O’Reilly (1992)

Manuale in linea di Linux: **man rpc**