



Università degli Studi di Bologna
Scuola di Ingegneria

Corso di Reti di Calcolatori T

Implementazione RPC: RPC di SUN

Antonio Corradi

Anno Accademico 2023/2024

RPC: MOTIVAZIONI E MODELLI

RPC – Cliente Servitore a livello applicativo con elevata trasparenza usando linguaggi diversi

La chiamata di procedura remota permette di invocare una procedura non locale disponibile su un nodo remoto

- L'operazione interessa un **nodo remoto** e consente ai clienti non locali di richiedere una funzione, **rimanendo a livello applicativo (formato dei dati diverso)**

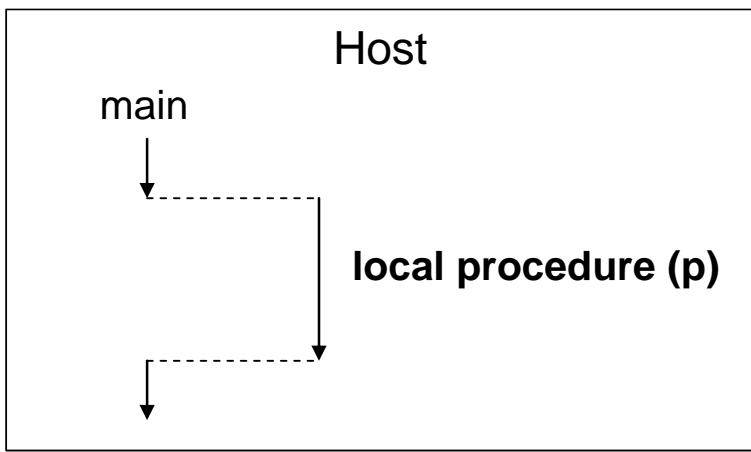
Modello di riferimento → CLIENTE/SERVITORE

invocazione di un **servizio remoto**
con **parametri con tipo e valore di ritorno**

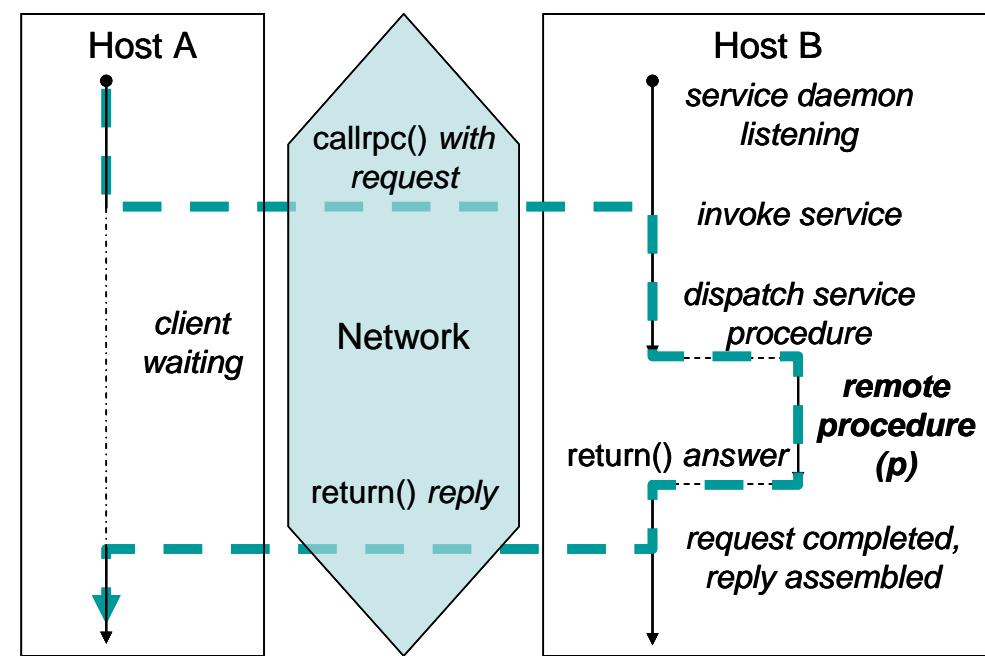
Modello di base: P.B. Hansen (Distributed Processing)
A. Birrel, B.J. Nelson

FUNZIONAMENTO INVOCAZIONE REMOTA

Invocazione **locale**
della procedura p



Invocazione **remota**
della procedura p



REQUISITI PER IMPLEMENTAZIONE RPC

Uso di una **infrastruttura di supporto** SUN che viene inglobata nei processi impegnati nella interazione

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** **default in SUN RPC**
- **at-most-once** **possibile in SUN RPC**
- **exactly-once**

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

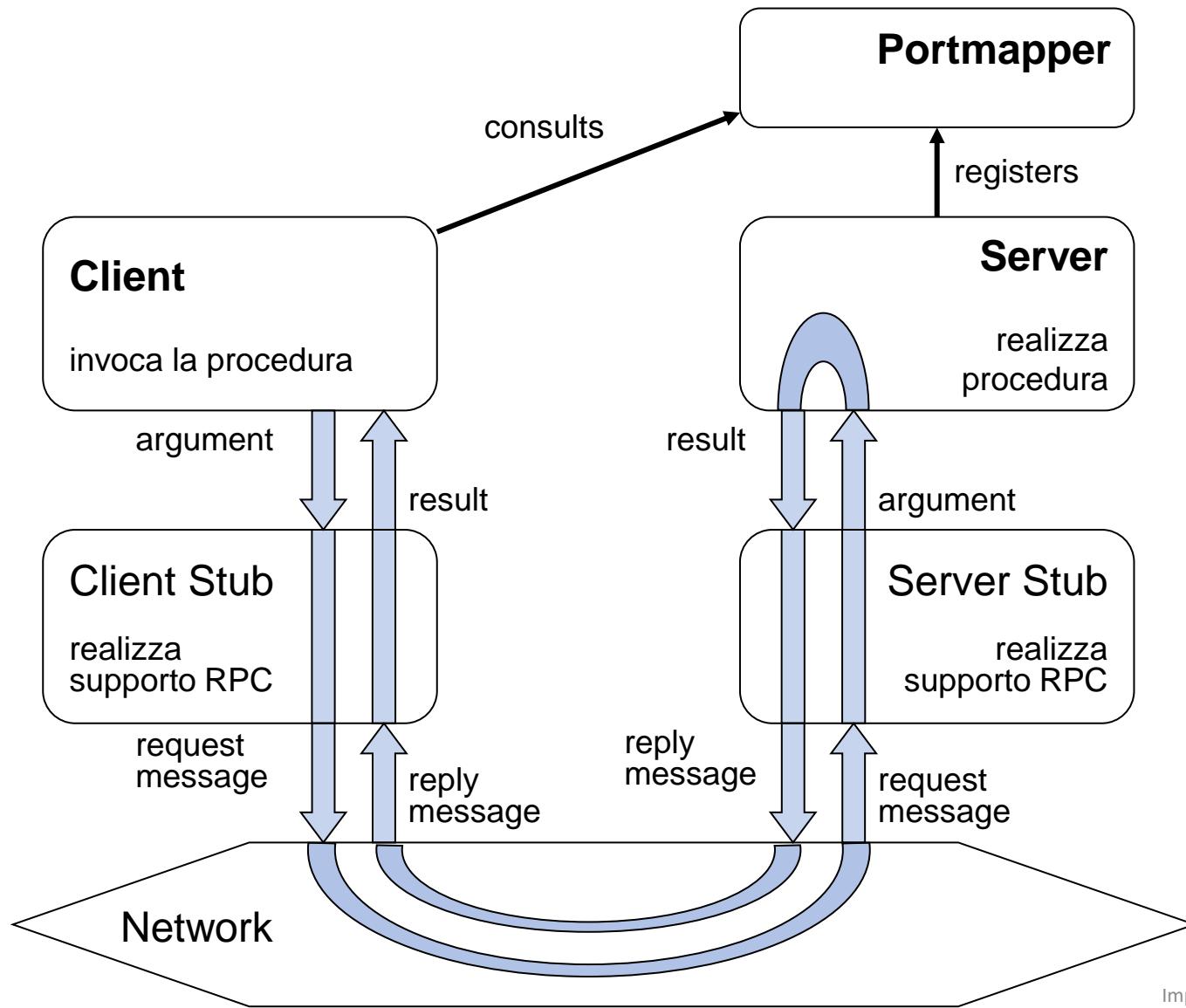
operazioni per il SERVITORE

- **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

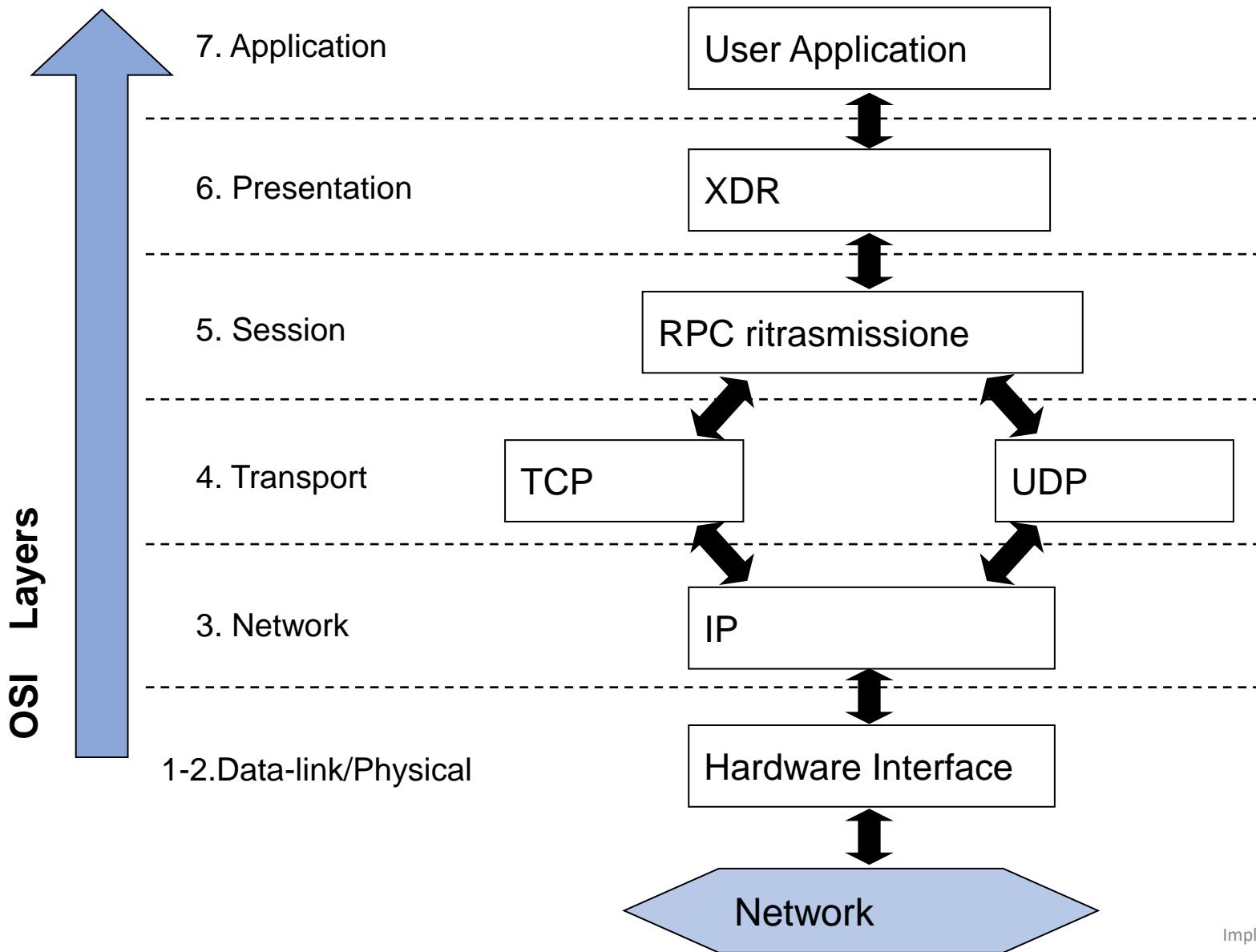
Portmapper

risoluzione indirizzo del servitore

Network File System (NFS)

file system distribuito di Sun

SUN RPC E STACK OSI



DEFINIZIONE DEL CONTRATTO RPC

Le RPC sono basate su un **contratto esplicito sulle informazioni scambiate** e che permetta un **accordo preciso tra un processo che si trova su una architettura diversa da quello del fornitore del servizio**

Ci sono due momenti

1) **Contratto tra le operazioni che si possono chiedere e fornire in base ad un sistema di nomi che tutti conoscono**
SUN sceglie di avere nomi basati sulla tripla

<#programma, #versione, #procedura>
e si tiene il protocollo come possibilità (UDP, TCP)

2) **Specificazione della procedura da invocare usando argomenti di tipo specificato:** si sceglie di avere un unico argomento in ingresso e un unico di uscita

Ricordiamo che siamo in architetture diverse e che i processi devono essere autocontenuti (no macchine virtuali)

DEFINIZIONE DEL PROGRAMMA RPC

Le **RPC** sono basate su un **contratto esplicito sulle informazioni scambiate**, che consiste di 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
cioè in formato XDR sorgente, che deve descrivere solo il contratto di interazione

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

DEFINIZIONE DEL SERVIZIO REMOTO

Diamo tutte le **definizioni per le procedure remote** nel file **stampa.x**

```
/* stampa.x file sorgente XDR */
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 un argomento di tipo string e un risultato di tipo int (intero)**

Si notino i vincoli per RPC di SUN:

- 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 cliente**: 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`

Mancano gli **stub cliente e servitore**, generati automaticamente per produrre i programmi **client e server** (e strutture interne – gestori TX, e funzioni interne di supporto RPC)

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



Chi invoca la procedura remota (lato server)?

RPC: UN PRIMO ESEMPIO

Servizio di stampa di messaggi su video 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 al servizio di stampa */
    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 rispetto al caso locale (in C)

- 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 **remoto** di **stampa a video**

```
#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 cliente cl che è un riferimento ad un elemento di supporto** (si veda oltre per ulteriori dettagli):

il **gestore di trasporto (cl)** gestisce la comunicazione col server, in questo caso a default 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)

VARIAZIONI RISPETTO AL CASO LOCALE

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 solo 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_pccreateerror** 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** (se non attivo)
 2. registrare i servizi presso il **Portmapper** (eseguendo il server)
 5. **Reperire** (lato **client**) **l'endpoint del server** tramite il Portmapper, **creando 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**

RPC E SISTEMI UNIX

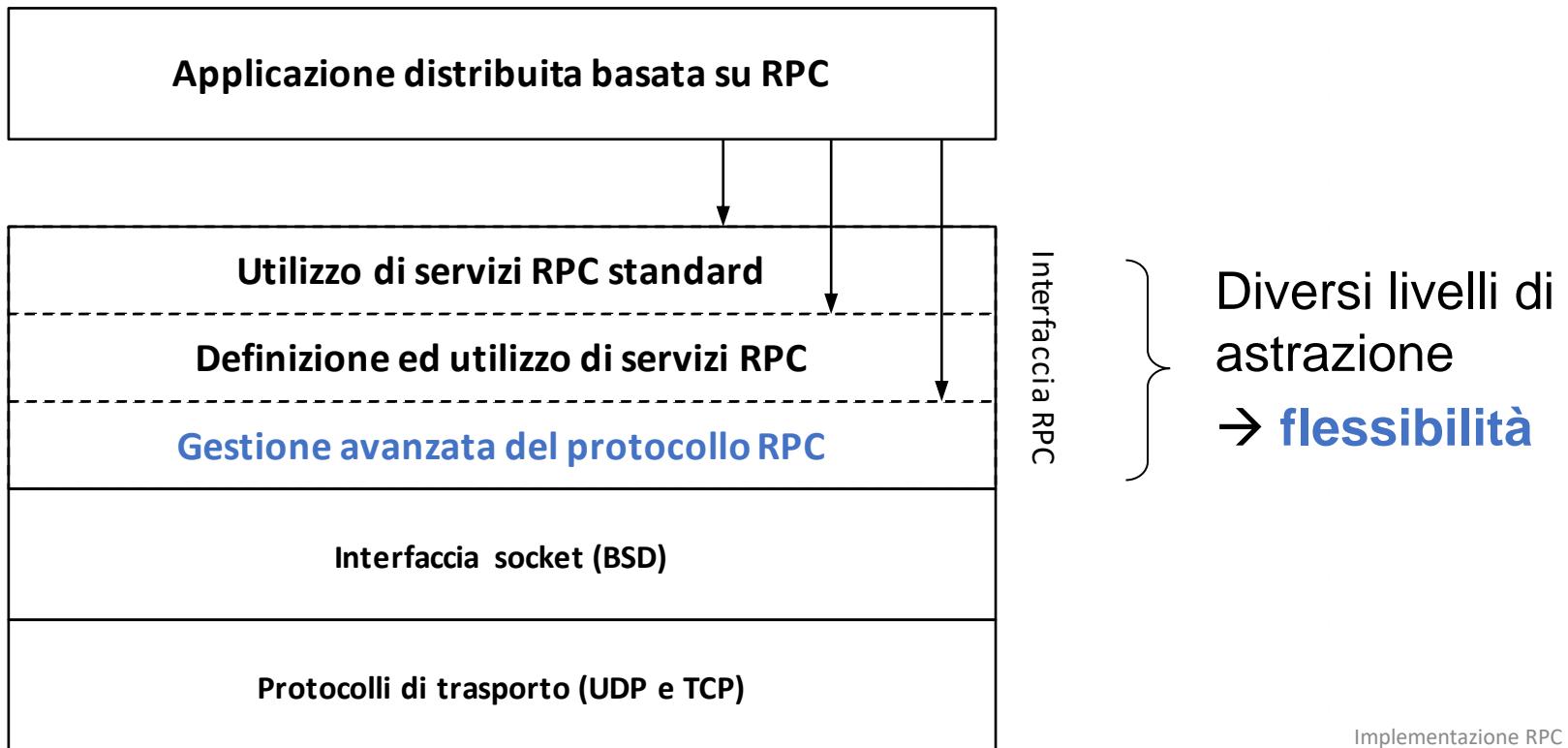
Ie RPC sfruttano i servizi delle socket:

sia UDP per datagrammi (**default SUN**)

- **servizi senza connessione**

sia TCP per stream

- **servizi con connessione**



SUN RPC

In caso di **RPC SUN** ci sono alcuni vincoli principali:

- Un **programma** tipicamente contiene **più procedure remote**
- Si prevedono 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 alla restituzione del risultato di ritorno al programma cliente, il **processo cliente** è **sospeso 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 insieme 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

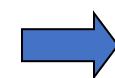
numero di programma
numero di versione
numero di procedura

per la identificazione globale

Numeri di programma in Hex (32 bit)

- 0 - 1fffffffh predefinito Sun → applicazioni d'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

SUPPORTO STANDARD RPC

Sun ha definito in modo preciso una serie di informazioni di supporto in XDR per le RPC (RFC relativo)

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

Vedi RFC



Unicità invocazione:

Ogni invocazione ha un XID proprio, ripetuto in caso di ritrasmissione

SUPPORTO STANDARD RPC

Sun ha definito in modo preciso i **corpi dei messaggi**

```
struct call_body {
    unsigned int rpcvers; /* must be equal to two (2) */
    unsigned int prog;
    unsigned int vers;
    unsigned int proc;
    opaque_auth cred; opaque_auth verf;
    /* procedure specific parameters start here */
};

union reply_body switch (reply_stat stat) {
    case MSG_ACCEPTED: accepted_reply areply;
    case MSG_DENIED: rejected_reply rreply;
} reply;
```

Chiamata di procedura XDR:

**Ogni invocazione ha un proprio sistema di nomi
completo: prog, versione e procedura**



SUPPORTO STANDARD RPC

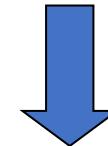
Sun ha definito in modo preciso il **sistema di nomi e anche una serie di informazioni di supporto** (usate per generare stub e consentire il massimo della operatività e **scambiare messaggi RPC**)

```
enum msg_type { CALL = 0, REPLY = 1};  
enum reply_stat {MSG_ACCEPTED = 0, MSG_DENIED = 1};  
  
enum accept_stat {SUCCESS=0, /* RPC execute with success */  
PROG_UNAVAIL = 1, /* remote hasn't exported program */  
PROG_MISMATCH = 2, /* remote can't support version # */  
PROC_UNAVAIL = 3, /* program can't support procedure */  
GARBAGE_ARGS = 4 /* procedure can't decode params */  
};  
enum reject_stat {  
RPC_MISMATCH = 0, /* RPC version number != 2 */  
AUTH_ERROR = 1 /* remote can't authenticate caller */  
};
```

ESITO DELLA CHIAMATA REMOTA

```
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 */
...};
```

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 RPC E SISTEMA DI NOMI

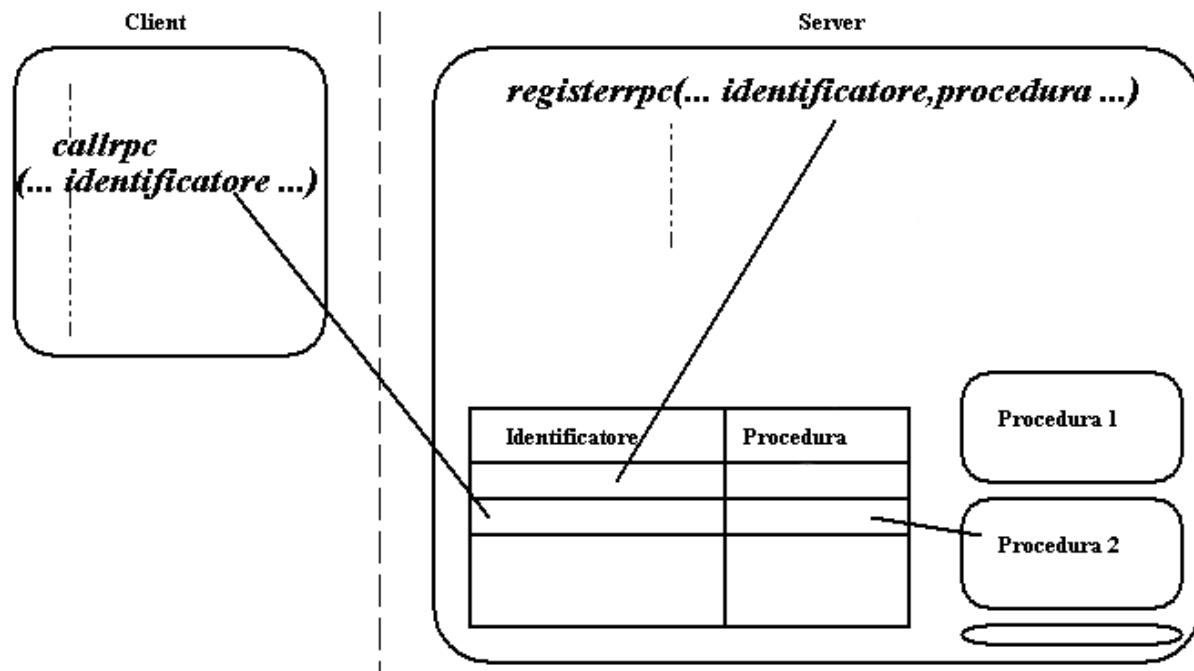
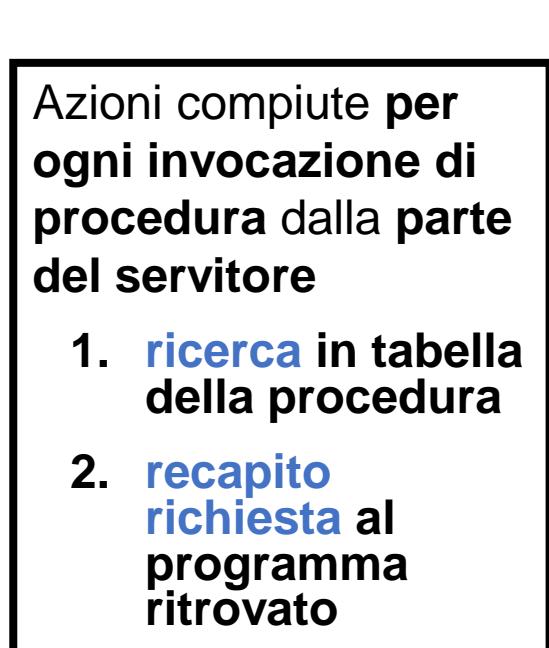
Per regolare e coordinare i servizi RPC sono necessari accordi tra clienti e servitori attraverso un portmapper

SERVER -
`registerrpc()`

registrazione: associa un identificatore unico alla procedura remota implementata nell'applicazione

CLIENT -
`callrpc()`

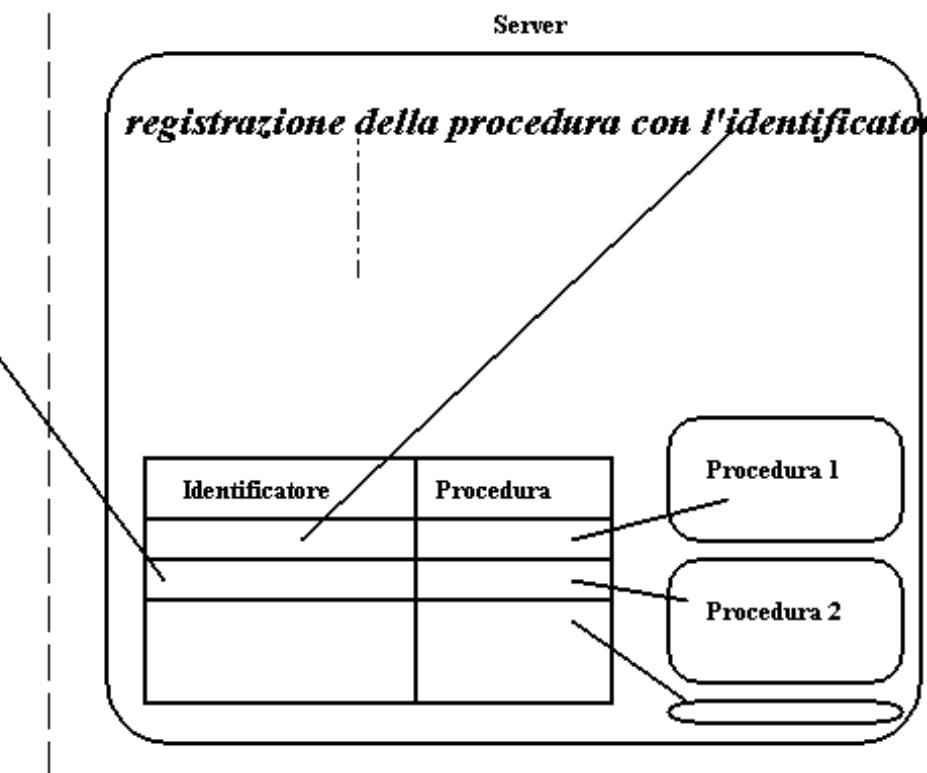
chiamata esplicita al meccanismo RPC con l'esecuzione della procedura remota



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 SVC_RUN

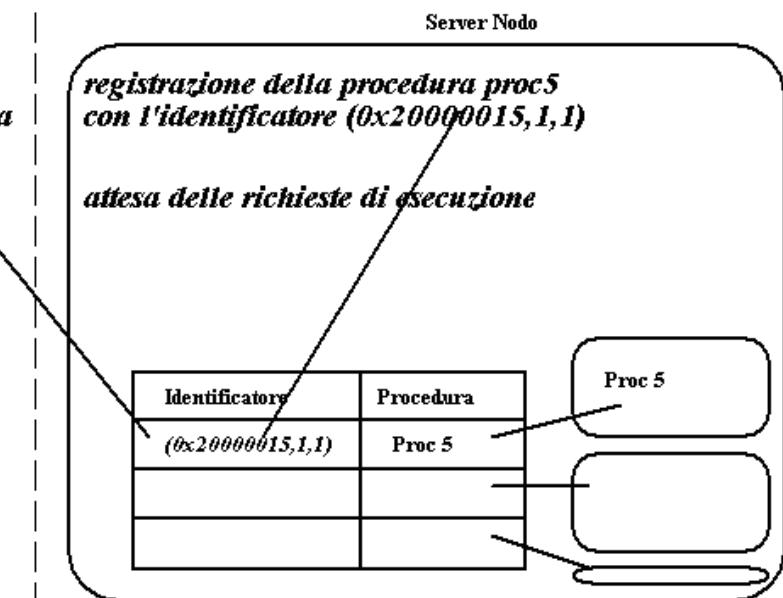
Dopo la registrazione della procedura, il processo non può terminare
Il processo che ha eseguito la registrazione deve **rimanere in attesa di chiamate ed essere risvegliato in caso di richiesta**

Uso della **svc_run()** con cui il processo diventa un demone in attesa di tutte le possibili

la **svc_run()** che esprime la attesa infinita del server e che non termina (è implementata come una select in un ciclo infinito)

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

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



I Processi Server attendono le richieste

CORRISPONDENZA DEI NOMI

In realtà, **il sistema di nomi portmapper si basa su una registrazione: tripla {numero_progr, numero_vers, protocollo_di_trasporto}** e **numero di porta {port}** in una tabella detta **port map**

Registrazione per programma, versione e protocollo

Manca il **numero di procedura dato** che tutti i servizi RPC (procedure all'interno di un programma) condividono lo stesso **gestore di trasporto** e sono **selezionati alla invocazione e non in fase di registrazione**

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
};

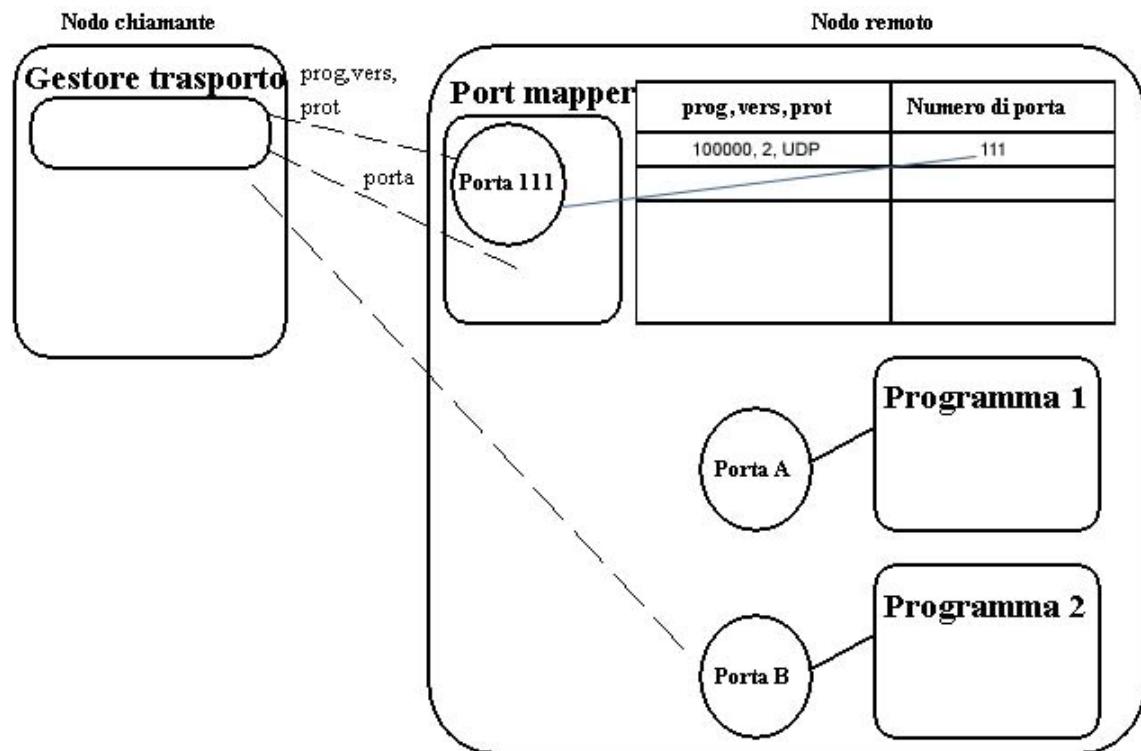
/* implementazione a lista dinamica della tabella */
struct *pmaplist { mapping map; pmaplist next; }
```

CORRISPONDENZA DEI NOMI

La **fase di chiamata** è preceduta da una **funzione di naming** con richiesta delle informazioni per il processo remoto (la **porta** a cui il processo è registrato)

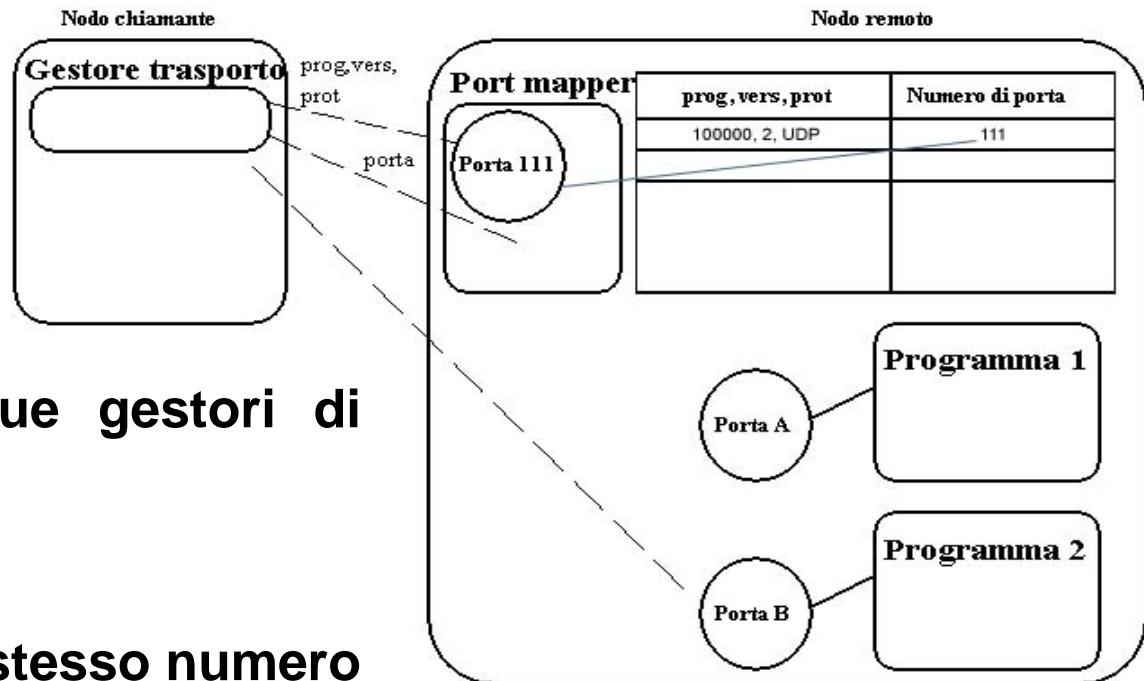
Questa informazione di stato è mantenuta in un **gestore di trasporto** del Cliente

Il portmapper registra il programma per economicità di registrazione e di contratto e non la procedura per disaccoppiare l'ottenimento del numero di porta per il servizio dalle singole chiamate



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 di **versione** del port mapper: **100000** **2**

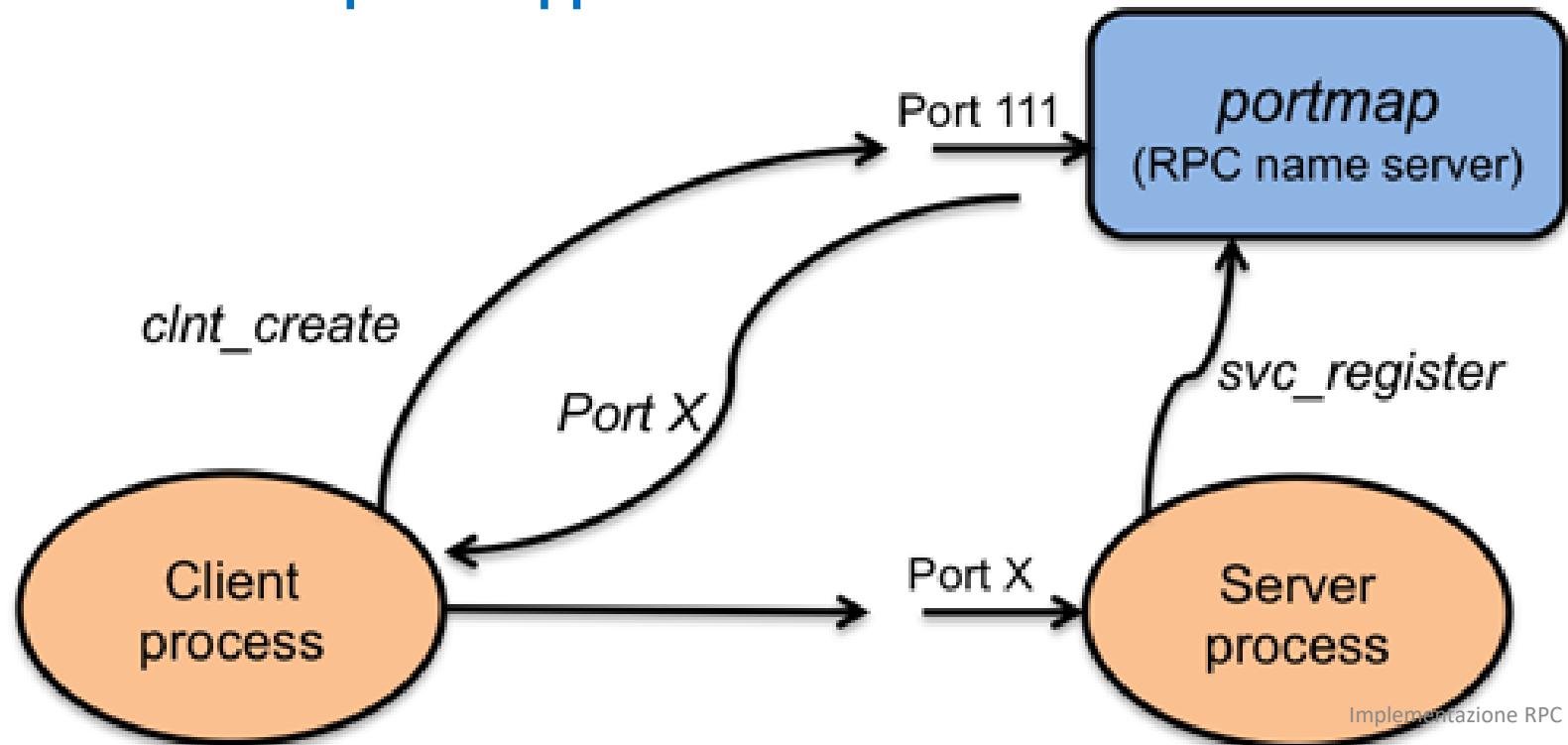
LIVELLO BASSO RPC

Tre attori che si devono coordinare

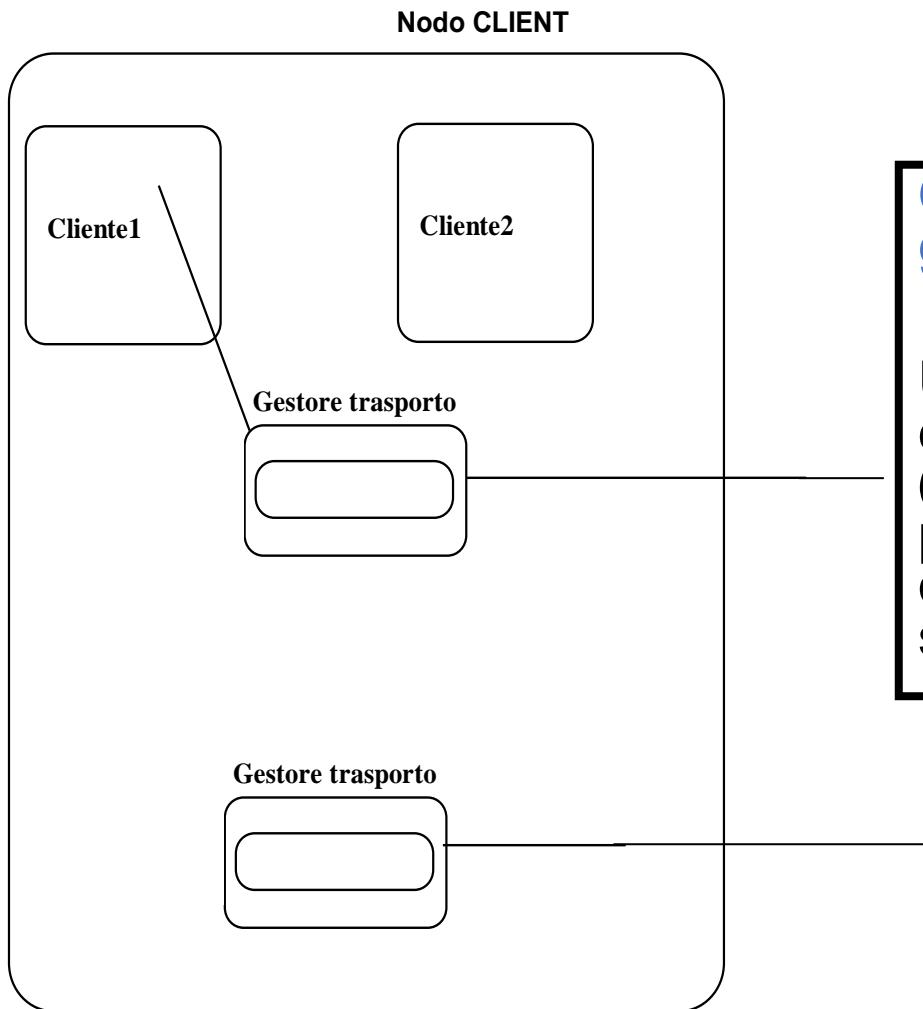
Il sistema di nomi, il **portmapper**, che riceve la registrazione dei server RPC locali e poi fornisce la porta per i clienti che fanno le richieste direttamente ai server per le procedure

Perché dividere il lavoro tra **cliente applicativo** e **stub**?

E il **cliente va lui al port mapper!!!!**



ARCHITETTURA E FUNZIONI RPC: LATO CLIENT



Creazione ed utilizzo di un gestore di trasporto lato client

Una **struttura dati** di supporto, detta **gestore di trasporto (CLIENT)**, viene usata nel **cliente** per tenere traccia e potere comunicare con i potenziali servitori

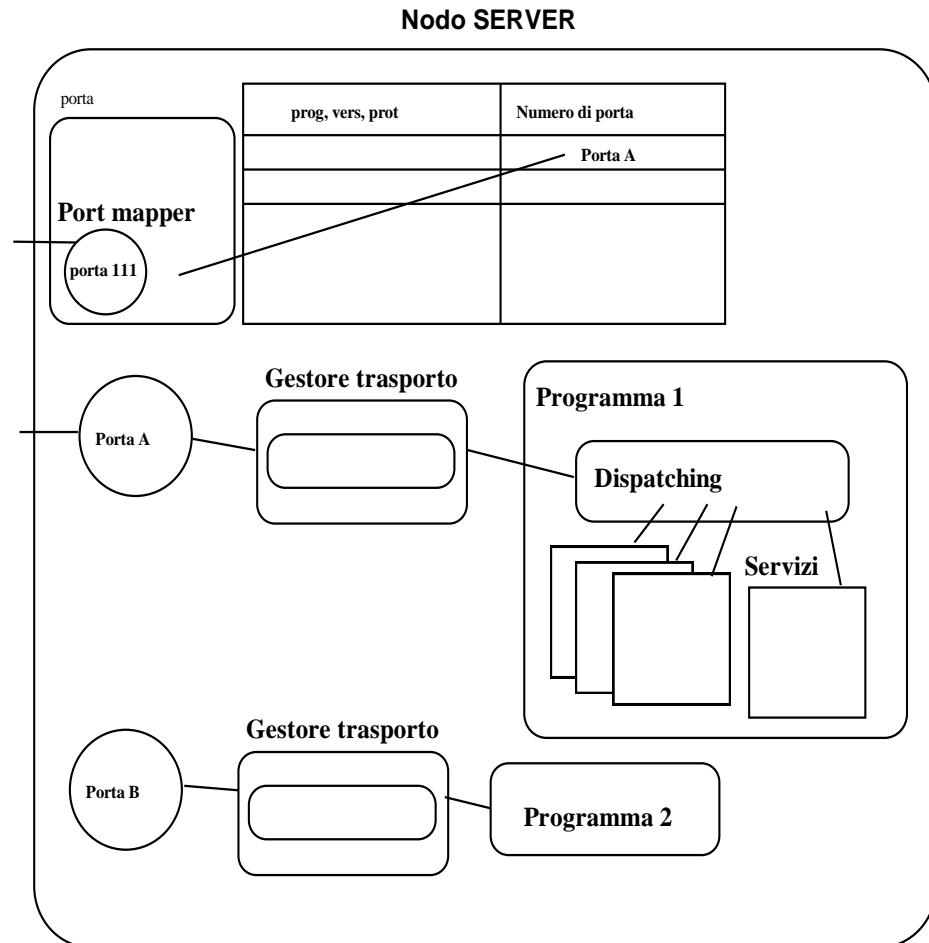
ARCHITETTURA E FUNZIONI RPC: LATO SERVER

Creazione e uso di un gestore di trasporto lato server

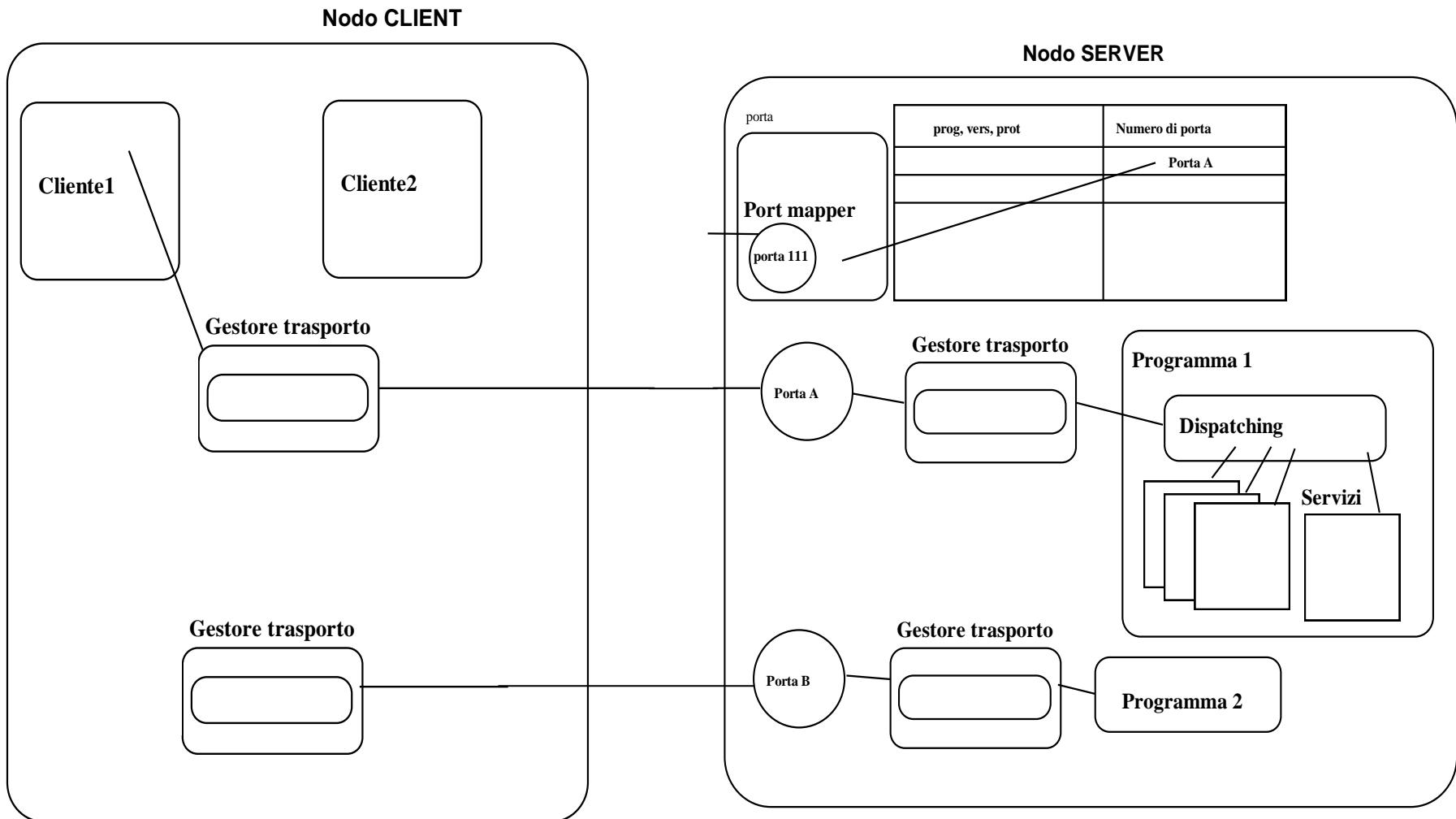
Il server usa il **gestore di trasporto servitore** per mantenere il collegamento RPC con i potenziali clienti, e in particolare con il servizio corrente

Dispatching (locale) del messaggio RPC

Per inoltrare la richiesta del client alla procedura corretta: il **messaggio RPC** contiene il **numero della procedura che viene identificato nella procedura di dispatching** quando attivata per ottenere il servizio



ARCHITETTURA E FUNZIONI RPC



PORT MAPPER: MOTIVAZIONI E DETTAGLI

Il PORT MAPPER è il **server di nomi**, lui stesso **server RPC**

Il **port mapper** identifica il numero di porta associato ad un qualsiasi programma → **allocazione dinamica dei servizi sui nodi**

Per limitare il numero di porte riservate →
un **solo processo** sulla porta **111**

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 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**

INTERFACE DEFINITION LANGUAGE

Per la descrizione dei dati parte del contratto tra un cliente e un servitore, si sono definiti dei linguaggi di Interfaccia, linguaggi special-purpose per determinare in modo non ambiguo le procedure ed i dati scambiati

eXternal Data Representation (o XDR) è

il linguaggio per le RPC di SUN

La interfaccia è costituita da procedure remote che accettano un solo un argomento ed un solo risultato → a volte necessità di definire una struttura che li raggruppa

- **Firma dei metodi**

Il parametro di ingresso e il risultato sono:

- Tipi **atomici predefiniti**
- Tipi **standard** (costruttori riconosciuti)
- Tipi **costruiti**

ESEMPIO COMPLETO DI UTILIZZO DI XDR

Esempio di file 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; };

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;
}=0x20000020;
```

EXTERNAL DATA REPRESENTATION (XRD)

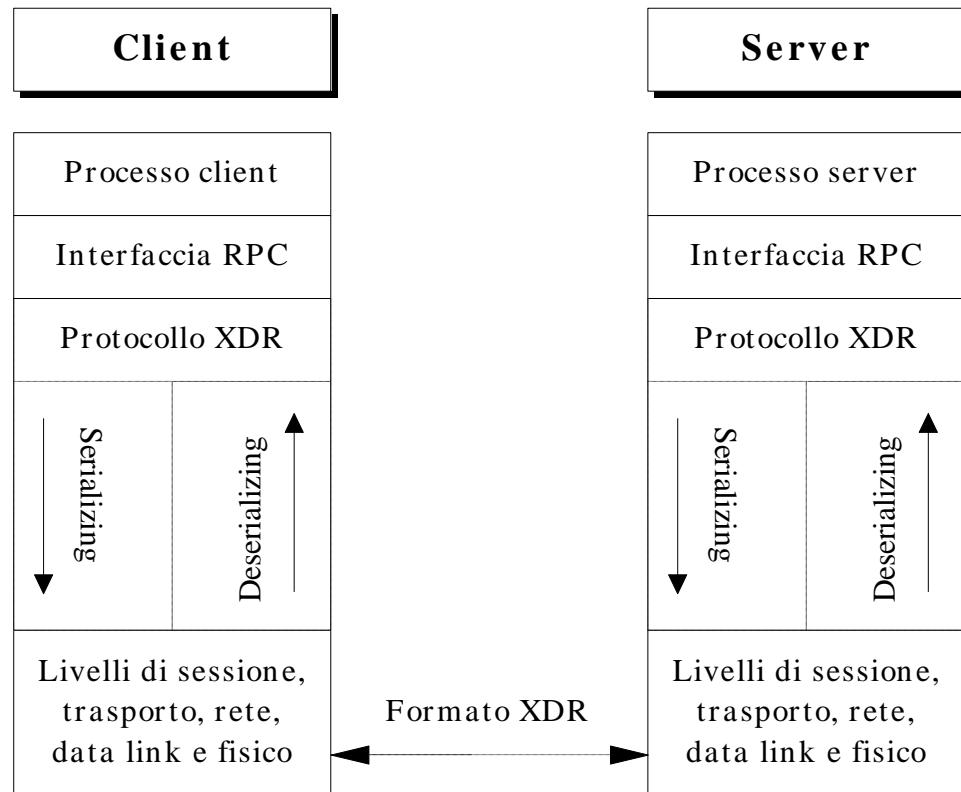
Funzioni svolte da protocollo XDR

→ **Marshalling**

che si colloca in OSI al livello di **Presentazione**

XDR procedure built-in di conversione, relative a:

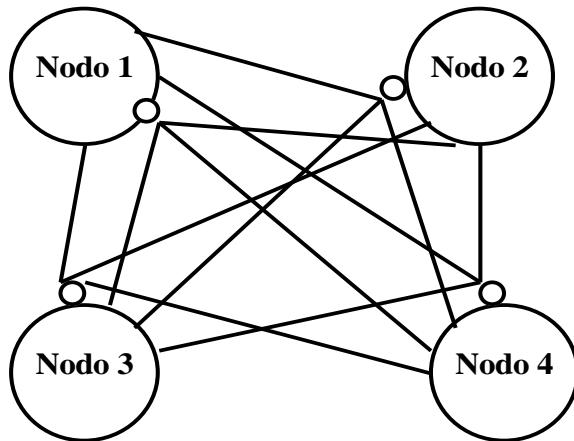
- Tipi **atomici predefiniti**
- Tipi **standard** (costruttori riconosciuti)



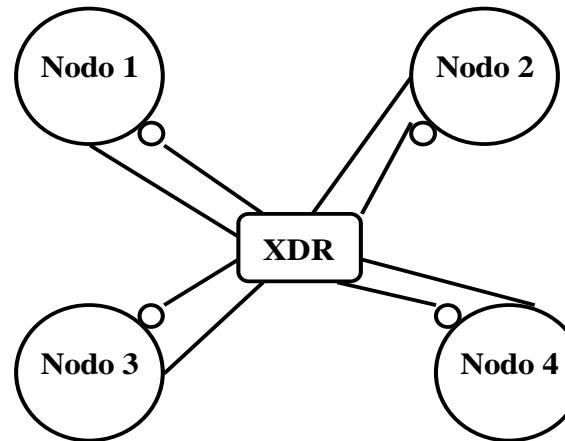
OMOGENEITÀ DEI DATI

Per comunicare tra **nodi eterogenei** due soluzioni

1. Dotare ogni nodo di **tutte le funzioni di conversione** possibili per ogni rappresentazione dei dati
 - **alta performance** e **alto numero** di funzioni di conversione → $N^*(N-1)$
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 → 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)

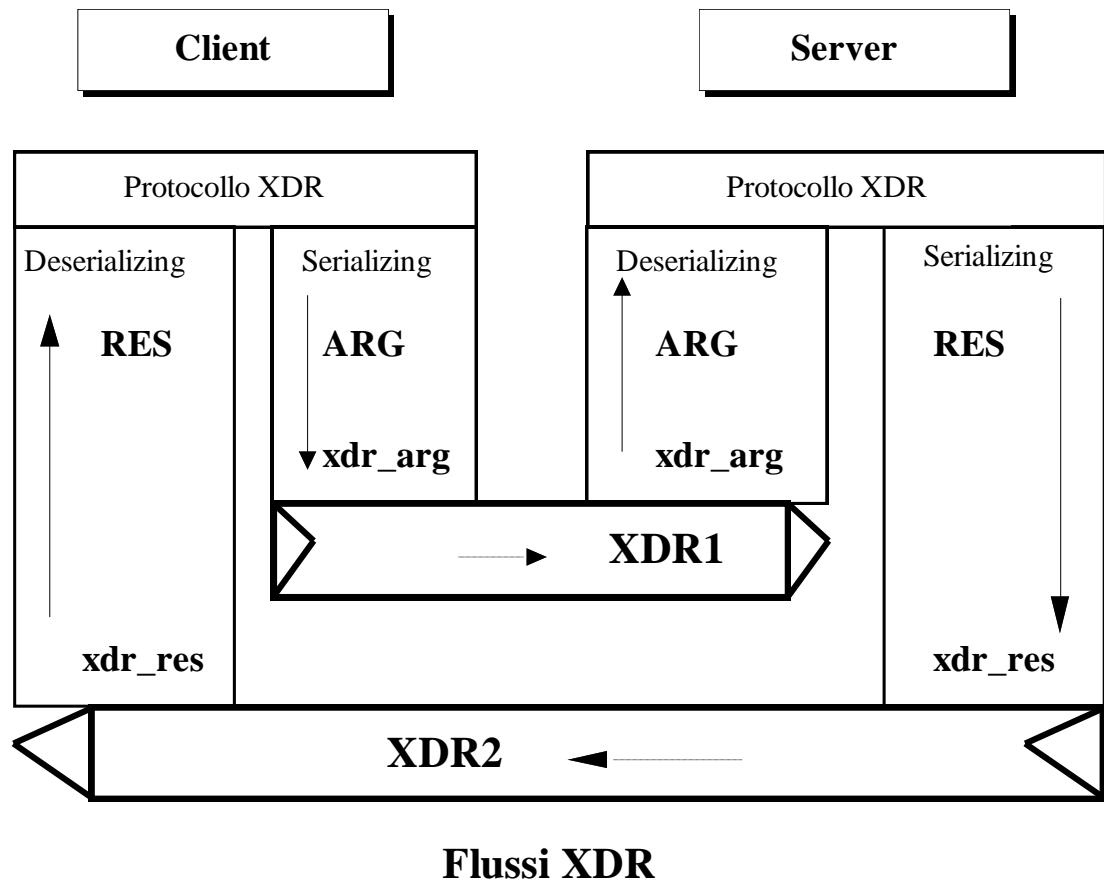
STREAM XDR

Per ogni informazione da trasmettere si hanno **due trasformazioni**, una al mittente e una al destinatario → sulla rete il **solo formato standard 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 la **sola trasformazione dei dati** (e non associate alla comunicazione)



FUNZIONI DI CONVERSIONE

Funzioni built-in

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

Funzioni per tipi composti

Funzione	Tipo di dato
<code>xdr_array()</code>	Vettori con elementi di tipo qualsiasi
<code>xdr_vector()</code>	Vettori a lunghezza fissa
<code>xdr_string()</code>	Sequenza di caratteri con terminatore a NULL <i>N.B.: stringa in C</i>
<code>xdr_bytes()</code>	Vettore di byte senza terminatore
<code>xdr_reference()</code>	Riferimento ad un dato
<code>xdr_pointer()</code>	Riferimento ad un dato, incluso NULL
<code>xdr_union()</code>	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 e hanno eseguito la operazione (da e per lo stream)

Possono funzionare anche con tipi definiti dall'utente, ad esempio **creazione di una struttura utente** 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);
}
```

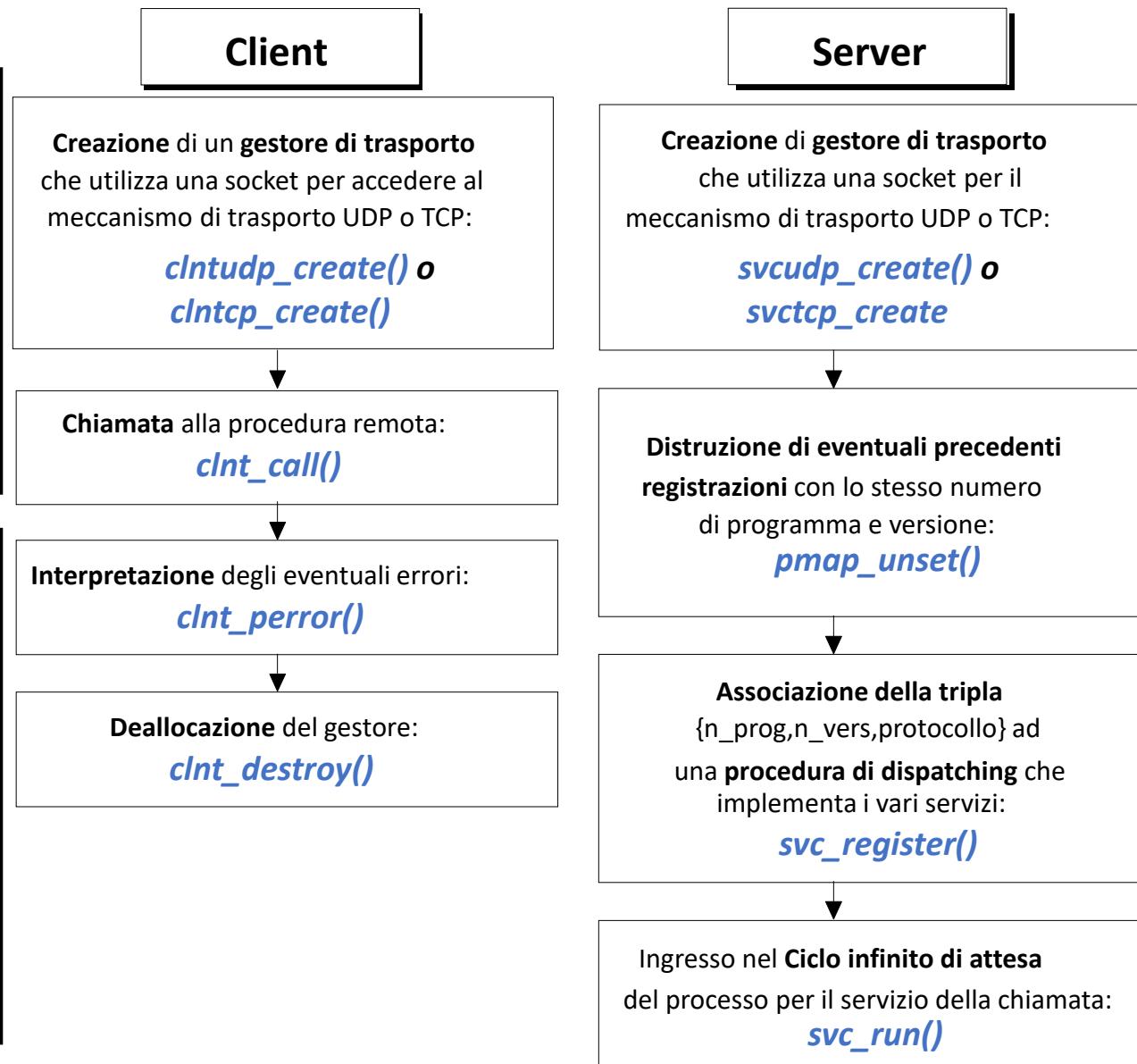
LIVELLO BASSO API RPC

Gestione avanzata del protocollo RPC

Le chiamate remote sono gestite 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



CREAZIONE DI UN GESTORE DI TRASPORTO SERVER

La `registerrpc()` utilizza

- `svcupd_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, struttura dati di supporto**, è definito dal tipo SVCXPRT

```
typedef struct {
#ifndef 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 xprt_stat (*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; caddr_t  xp_p2;    /* privato */
} SVCXPRT;
```

SERVER: GESTORE DI TRASPORTO (ANCORA)

SVCXPRT 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;
```

SERVER: GESTORE DI TRASPORTO (ANCORA)

svcupd_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** deve essere registrata preliminarmente per essere invocata alla chiamata

La procedura deve sistemare i parametri e deve trovare la procedura sul server, invocarla, trattare il risultato, e consegnarlo, terminando

La API **svc_register** associa numero programma e versione alla procedura di 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, proignum, versnum, dispatch,  
protocol)
```

```
SVCXPRT *xprt; /* gestore di trasporto */  
u_long proignum, versnum, protocol;  
char (*dispatch());  
/* funzione di selezione attivata alla call */
```

SERVER: PROCEDURA DI DISPATCHING

xptr	⇒ gestore di trasporto
proignum, 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?
Vedi stub tipico**

SERVER: INFORMAZIONI DI SUPPORTO

La procedura di **dispatching** contiene i **riferimenti alle implementazioni dei servizi** di un programma RPC → **stesso gestore e 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 a sola lettura */
    SVCXPRT * rq_xprt;         /* gestore associato */
};
```

SERVER: INFORMAZIONI DI SUPPORTO

```
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: PROCEDURE DI CONVERSIONE

```
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

SERVER: PROCEDURE DI CONVERSIONE

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

CLIENT: CREAZIONE DI UN GESTORE DI TRASPORTO

Il client **necessita di un gestore di trasporto** per RPC

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

Anche **clnttcp_create()** per protocollo affidabile

La **callrpc()** ottiene un gestore di trasporto con **clntudp_create()**

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: CREAZIONI DI UN GESTORE DI TRASPORTO

```
CLIENT * clntudp_create(addr, proignum, versnum,  
                        wait, sockp)
```

```
    struct sockaddr_in *addr;
```

```
    u_long proignum,versnum;
```

```
    struct timeval wait; int *sockp;
```

```
CLIENT * clnttcp_create(addr, proignum, versnum,  
                        sockp, sendsz, recvsz)
```

```
    struct sockaddr_in * addr; u_long proignum,versnum;
```

```
    int * sockp; u_int sendsz,recvsz;
```

wait ⇒ durata dell'intervallo di time-out

sendsz, recsz ⇒ dimensione dei buffer

CREAZIONE DI UN GESTORE DI TRASPORTO CLIENTE

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



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

CLIENT: CHIAMATA DELLA PROCEDURA REMOTA

Creata 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; xdrproc_t outproc; /* routine XDR */  
char *in; char *out; struct timeval tout;
```

CLIENT: CHIAMATA DELLA PROCEDURA REMOTA

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

→ **Più gestori** possono condividere una **stessa socket**

```
void clnt_destroy (clnt)
```

```
    CLIENT *clnt;
```

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 COMPLETO DI UTILIZZO DI XDR

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;
    }=0x20000020;
```

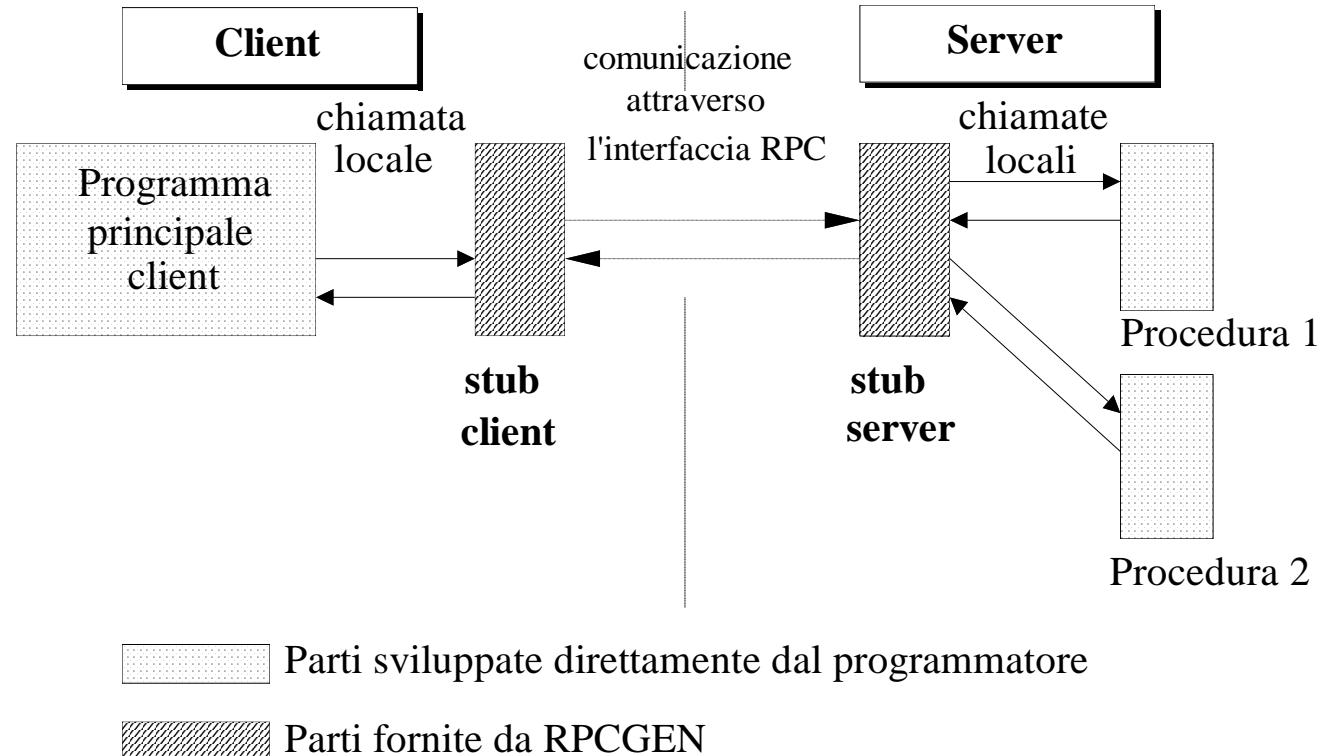
Generazione Automatica delle RPC

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

Remote Procedure Call Generator (rpcgen)

compilatore di protocollo RPC che genera procedure stub in modo automatico (e tutto il necessario)

RPCGEN processa
un insieme di
costrutti descrittivi
per tipi di dati e per
le procedure remote
→ **linguaggio XDR**



PROCESSO DI SVILUPPO

Data una **specific**a di partenza

- file di linguaggio XDR → `esempio.x`

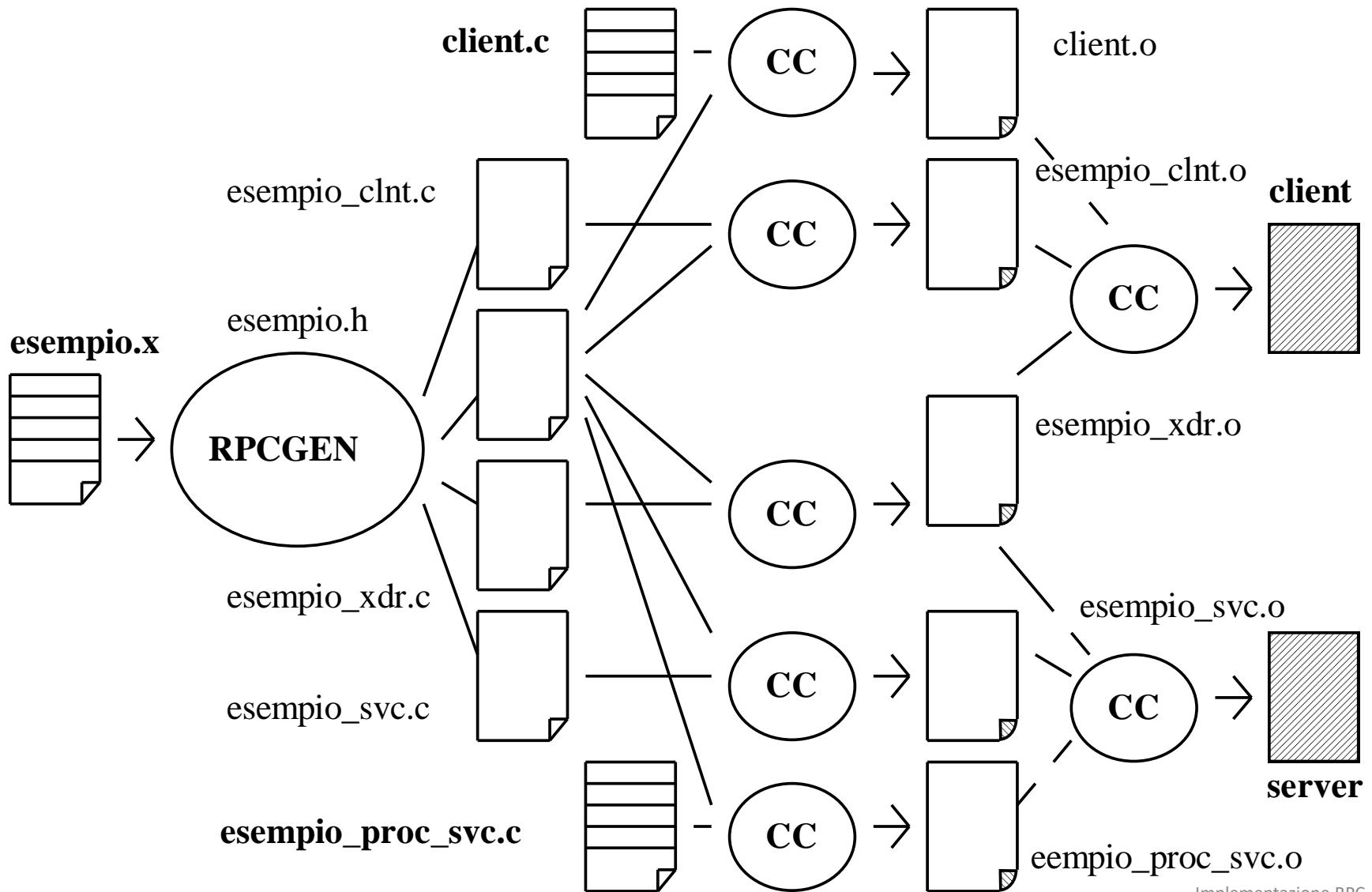
RPCGEN produce **in automatico**

- 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`

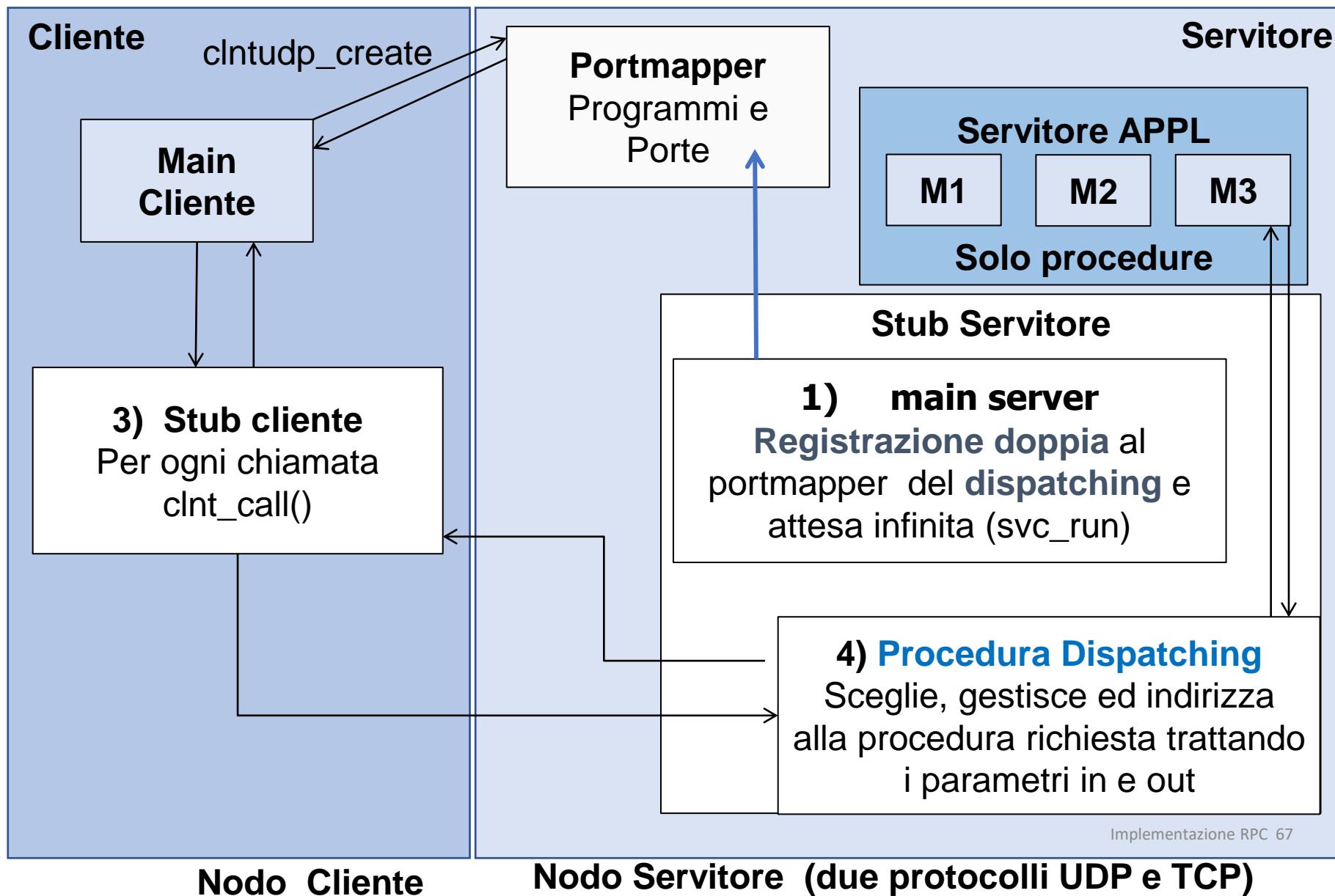
Lo **sviluppatore** deve realizzare

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

PROCESSO DI SVILUPPO



CONTENUTO DEI COMPONENTI C/S



ESEMPIO COMPLETO: FILE .X

Servizio di **remote directory list (RLS)**

una procedura remota che fornisce la lista dei file 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; };
```

ESEMPIO COMPLETO: FILE .X

```
const MAXNAMELEN = 255;
/* argomento della chiamata e... ricapitolo
typedef string nametype <MAXNAMELEN>;
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;
```

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 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>
#ifndef __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

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

```
/* rls.h */
#include <rpc/rpc.h>
#ifndef __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    #define REaddir 1
#if defined(__STDC__) || defined(__cplusplus)
```

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 */
#endif /* __cplusplus
```

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 RPCGEN PER TIPI UTENTE: 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; }
```

FILE RPCGEN PER TIPI UTENTE: RLS_XDR.C

```
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 DI 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;
/* deregistrazione 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);}
```

FILE PRODOTTI DA RPCGEN: STUB SERVER

```
/* 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 */
/* qui non si arriva */
fprintf (stderr, "%s", "svc_run returned"); exit (1);
}
```

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;
}
```

FILE PRODOTTI DA RPCGEN: STUB SERVER

```
/* rls_svc.c (stub del server): dopo configurazione
procedura di dispatching - invocazione chiamata locale*/
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 **rIs_clnt.c** offre al client **la procedura readdir_1()**

stub **rIs_svc.c** contiene la **registrazione dei servizi in RPC**, sia come servizi TCP che come servizi UDP e la **procedura di dispatching rIsprog_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); }
  /* altrimenti 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;
/* si libera la memoria della chiamata precedente */
xdr_free((xdrproc_t) xdr_readdir_res, (caddr_t) &res);
/* apertura di una directory */
dirp = opendir(*dirname);
if( dirp==NULL ) {res.remoteErrno=errno; return &res;}
nlp = &res.readdir_res_u.list;
```

A cosa serve **rd**? Si riprenda la definizione della struttura **svc_req** sopra

FILE SVILUPPATI DAL PROGRAMMATORE: SERVER

```
/* file rls_svc_proc.c: corpo programma server */
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;
}
/* chiusura della lista con un puntatore a NULL */
nlp=NULL;
res.remoteErrno=0;
/* chiusura directory */
closedir(dirp);
return &res;
}
```

Ancora a cosa serve **rd**? Si riprenda la definizione della struttura **svc_req** 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

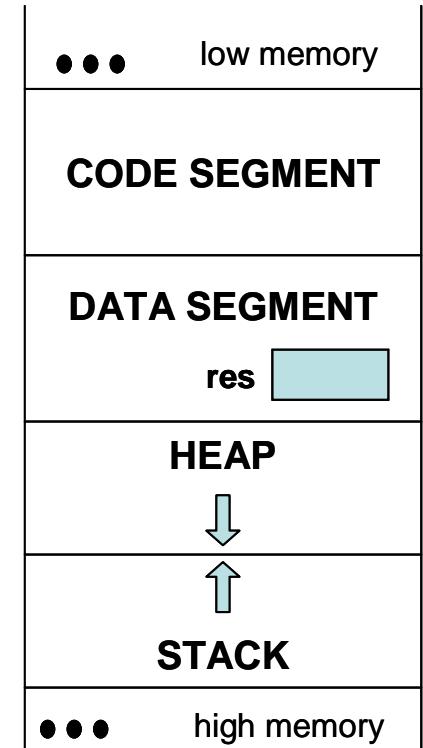
Tempo di vita

- 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 al termine della procedura, res verrebbe **rimosso dallo stack**

E quando allochiamo noi la struttura dati (malloc)?



ANCORA DETTAGLI SU XDR

XDR non è un linguaggio di programmazione ma **un linguaggio di descrizione delle procedure ed i dati** (detto spesso **Linguaggio di Definizione dei dati di Interfaccia**, o **Interface Definition Language**, o **IDL**)

- Non viene supportato da un vero compilatore
- Non è preciso nella traduzione
- Potrebbe obbligare alla ridefinizione delle strutture in passi per ottenere la espansione...
- In ogni caso, se non ha successo, potrebbe non espandere niente e lasciare il compito al compilatore vero 😞
- **Bisogna sempre controllare il sorgente espanso**

FUNZIONI DI CONVERSIONE

Funzioni built-in

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

Funzioni per tipi composti

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

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 { usigned 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 i due stub e i programmi addizionali

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 byte 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 sempre il controllo di tipo

Se il tipo indicato non appartiene a ai tipi riconosciuti, il compilatore RPCGEN assume che sia definito a parte →

Ossia che le funzioni di conversione XDR siano 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** e produce le funzioni di trasformazione corrette

```
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 (una sequenza di valori)**

Cambia la **funzione xdr di conversione che controlla la lunghezza**

ESEMPIO ARRAY CON MEMORIA DINAMICA

```
#define LUNGFILe 100
#define MAXLIST 6
typedef string nomefile <LUNGFILe>
typedef nomefile listafile <MAXLIST>
```

Qui avremo due **strutture dinamiche** che richiedono di trattare la allocazioine della memoria per il livello della lista e anche per ogni elemento

La seconda richiede espansione in **listafile_len** e **listafile_val**

- il primo contiene il numero di **posizioni occupate**
- il secondo è un **puntatore ad un vettore con i dati (una sequenza di valori)**

Dovremo allocare con una malloc la variabile **lista** e tutti i nomefile

```
lista.listafile_val= malloc(sizeof (listafile));
for (i=0;i<MAXLIST;i++) lista_val[i]= malloc(sizeof(nomefile));
lista.listafile_len=MAXLIST
```

Il singolo elemento anche

```
& lista.listafile_val [i] [0]
```

ESEMPIO ARRAY CON MEMORIA STATICÀ

Si può lavorare con strutture staticamente definite

```
#define LUNFILE 100
#define MAXLIST 6
struct nomefile {char nome [LUNFILE] };
struct listofile {nomefile lis [MAXLIST] };
listofile l;
```

Lista matrice pronta senza allocare memoria

l. lis. nome [i]

Non sono possibili scorciatoie notazionali in XDR

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 "}"
```

case-list:

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

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 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>

DICHIARAZIONE DI TIPI COSTANTE

Costanti simboliche

const-definition:

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

Ad esempio, nella **specificazione 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
<code>typedef string fname_type <255>;</code>	<code>typedef char *fname_type;</code>

ANCORA RPC: MODALITÀ ASINCRONA

A default 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 deve prevedere 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;

MODALITÀ ASINCRONA (ANCORA)

`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

Il servitore ed il cliente basano la fiducia che le cose funzioni non bene
non sul **livello applicativo (non cliente/servitore sincrono)** ma sul
trasporto affidabile

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

MODALITÀ ASINCRONA BATCH

Tutte le richieste di servizio del client **vengono poste nel buffer TCP** e gestite dalla driver di trasporto **senza bloccare il processo che le genera**

Modalità batch

- ogni chiamata non richiede risposta e ogni servizio non invia risultati
- le richieste sono trasportate con un protocollo affidabile (come TCP)

Implementativamente:

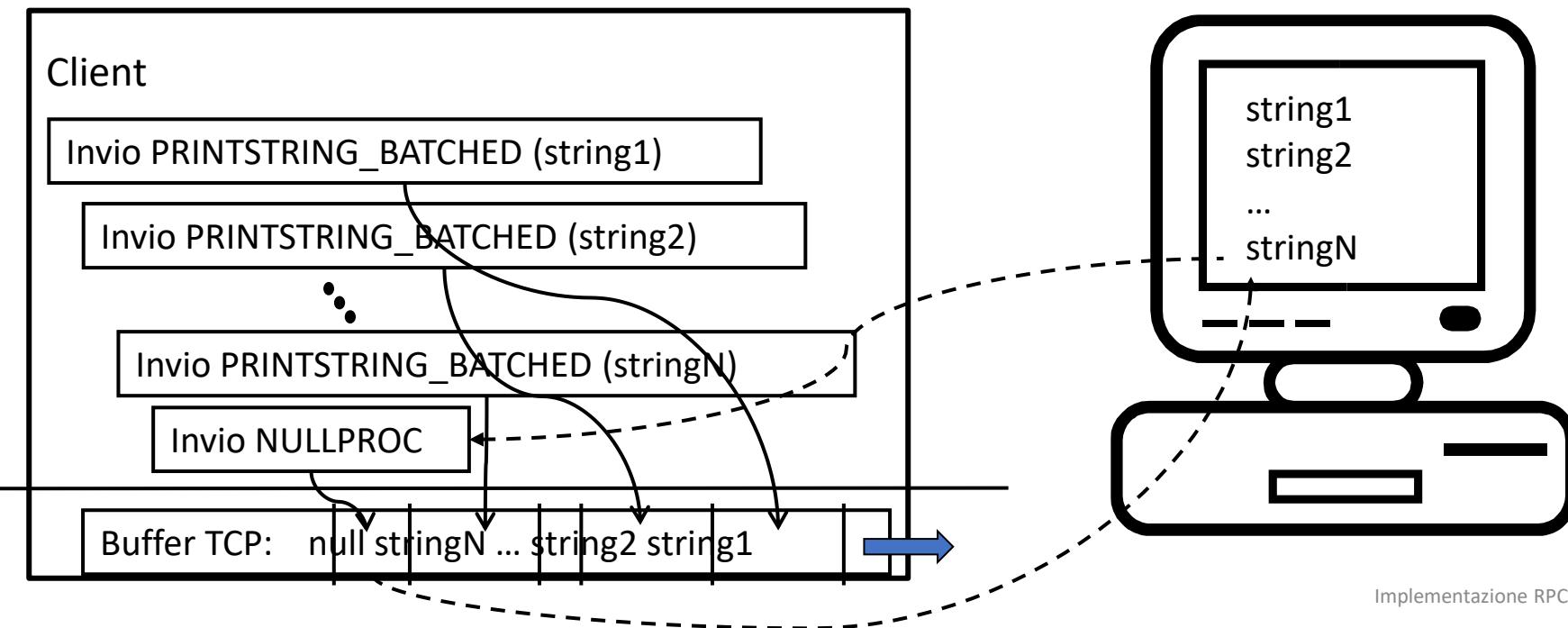
- impiego del protocollo **TCP**
- valore **nullo del timeout** nella primitiva **clnt_call()**

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 DI RPC ASINCRONA

Una serie di **chiamate asincrone** per la stampa di stringhe (procedura **PRINTSTRING_BATCHED**) sul nodo remoto: si termina con **una chiamata sincrona** alla procedura nulla (**NULLPROC**) che blocca il client fino alla ricezione dell'avvenuta esecuzione della richiesta di NULLPROC accodata dopo le richieste di PRINTSTRING_BATCHED



CLIENT 1/3

```
#define PROG (unsigned long) 0x20000020
#define VERS (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);}
```

CLIENT 2/3

```
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;
/* gestore TCP */
if((client=clnttcp_create(&server_addr, PROG, VERS,
                           &sock, 0, 0))==NULL)
{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 e spedizioni
asincrone al nodo remoto, fino a EOF */
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);}
}
```

CLIENT 3/3

```
/* Ultima chiamata RPC sincrona per svuotare completamente
   il buffer TCP */
total_timeout.tv_sec=20;      /* NB: timeout non nullo!! */

/* Chiamata sincrona di NULLPROC */
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); }

/* libero le risorse usate */
clnt_destroy(client);

}
```

SERVER 1/3

```
#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");
}
```

SERVER 2/3

```
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_BATCHED:
    if (!svc_getargs(transp, xdr_wrapstring, &s) )
    {fprintf(stderr, "problemi decodificare argomenti.\n");
     break; }
    fprintf(stderr,"%s\n",s);
    svc_freeargs(transp,xdr_wrapstring,&s); // vedi dopo
/* questo è un servizio asincrono: non c'è svc_sendreply() */
    break;
default:
    svcerr_noproc(transp); return;
}
```

SERVER 3/3

```
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 senza dover attendere conferme dal server (**interazione asincrona**)

Lo svuotamento della buffer TCP avviene sicuramente all'invocazione della NULLPROC (**sincrona**)

BIBLIOGRAFIA

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

Manuale in linea di Linux per parte di supporto API:

man rpc

Vedere anche tutta la documentazione Oracle di RPC

https://docs.oracle.com/cd/E18752_01/html/816-1435/rpcproto-13077.html

Anche XDR

https://docs.oracle.com/cd/E18752_01/html/816-1435/xdrproto-61652.html