Università degli Studi di Bologna
Scuola di Ingegneria

Corso di
Reti di Calcolatori T

Esercitazione 8 (svolta)
Remote Procedure Call – RPC

Antonio Corradi, Luca Foschini
Michele Solimando, Giuseppe Martuscelli, Marco Torello
Anno accademico 2020/2021

SPECIFICA: ESERCIZIO 1

Sviluppare un'applicazione C/S che consente di **effettuare la somma e la moltiplicazione tra due interi in remoto**

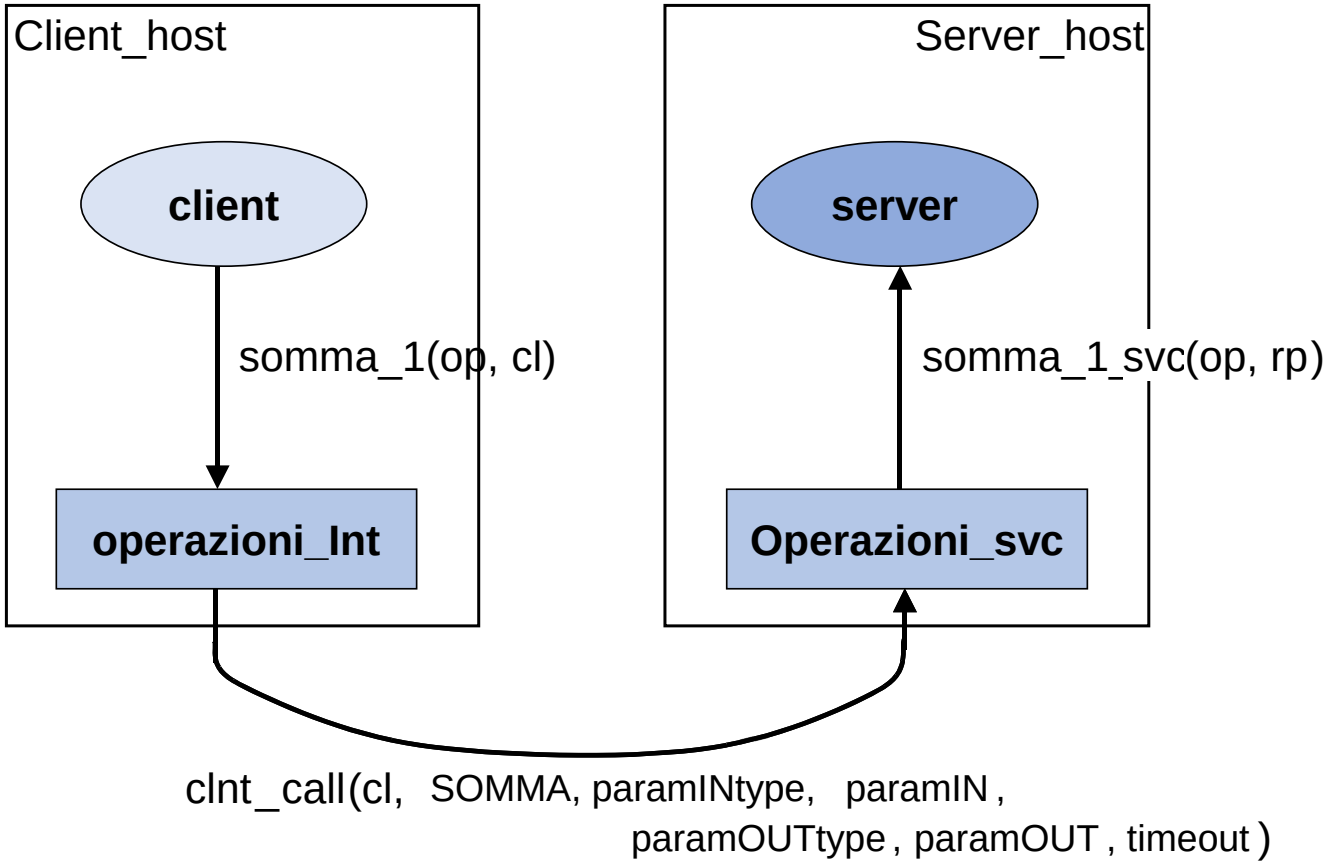
Il **Client** viene invocato da linea di comando con tre argomenti, i.e., il tipo di operazione e due interi che si vogliono sommare o moltiplicare, esegue la chiamata alla **procedura remota richiesta** passando gli operandi nella struttura dati **Operandi**, e stampa il risultato dell'operazione (**int**) ottenuto come valore di ritorno della procedura

Il **Server** esegue la procedura che effettua la **somma** o la **moltiplicazione** tra i due parametri e restituisce il risultato dell'operazione al client

Variante: realizzare il client ciclico, che chiede tipo di operazione e operandi all'utente da console fino a fine file

ARCHITETTURA

Esempio per **somma**



FILE OPERAZIONI.X

```
struct Operandi{ int op1; int op2; };

program OPERAZIONIPROG {
  version OPERAZIONIVERS {
    int SOMMA(Operandi) = 1;
    int MOLTIPLICAZIONE(Operandi) = 2;
  } = 1;
} = 0x20000013;
```

Compilazione per generare il file header, il file per le conversioni xdr e gli stub:

rpcgen operazioni.x

Produce i file:

- **operazioni.h** da includere in **operazioni_proc.c** e in **operazioni_client.c**
- **operazioni_xdr.c** che contiene le routine di conversione xdr
- **operazioni_clnt.c** stub del client
- **operazioni_svc.c** stub del server

FILE OPERAZIONI.H 1/2

```
/* Please do not edit this file. It was generated using rpcgen. */
#ifndef _OPERAZIONI_H_RPCGEN
#define _OPERAZIONI_H_RPCGEN
#include <rpc/rpc.h>
#ifdef __cplusplus
extern "C" {
#endif

struct Operandi { int op1; int op2; };
typedef struct Operandi Operandi;

#define OPERAZIONIPROG 0x20000013
#define OPERAZIONIVERS 1
/*ANSI C*/
#if defined(__STDC__)||defined(__cplusplus)
#define SOMMA 1
extern int * somma_1(Operandi *, CLIENT *);
extern int * somma_1_svc(Operandi *,struct svc_req*);
#define MOLTIPLICAZIONE 2
extern int * moltiplicazione_1(Operandi *, CLIENT *);
extern int * moltiplicazione_1_svc(Operandi *,struct svc_req*);
extern int operazioniprog_1_freeresult (SVCXPRT *,
  xdrproc_t, caddr_t);

/* K&R C */
// vedere file… stesse definizioni in formato C Kernigham - Ritchie
```

FILE OPERAZIONI.H 2/2

```
/* the xdr functions */
#if defined(__STDC__) || defined(__cplusplus)
extern  bool_t xdr_Operandi (XDR *, Operandi*);

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

#ifdef __cplusplus
}
#endif

#endif /* !_OPERAZIONI_H_RPCGEN */
```

Il file header contiene **tutte le dichiarazioni** che possono essere necessarie a un componente e sono anche messe a disposizione nei **due formati C standard** e **C Kernigham - Ritchie**

FILE OPERAZIONI_XDR.C: ROUTINE DI CONVERSIONE

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

```
#include "operazioni.h"

bool_t  xdr_Operandi (XDR *xdrs, Operandi *objp)
{register int32_t *buf;

    if (!xdr_int (xdrs, &objp->op1))
       return FALSE;
    if (!xdr_int (xdrs, &objp->op2))
       return FALSE;
  return TRUE;
}
```

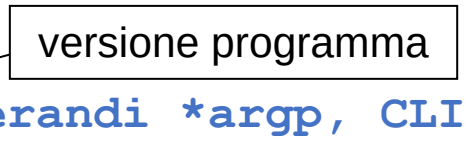
I file XDR sono tutti **insiemi di funzioni** per la **trasformazione dei dati dalla rappresentazione locale alla standard** (XDR) e **viceversa**
Ogni funzione è ottenuta scandendo **la sequenza dei dati** a ricorrendo alla **funzione XDR dei parametri elementari**

FILE OPERAZIONI_CLNT.C: STUB DEL CLIENT 1/2

```
/* Please do not edit this file. It was generated using rpcgen. */
#include <memory.h> /* for memset */
#include "operazioni.h"

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

int * somma_1(Operandi *argp, CLIENT *clnt)
{ static int clnt_res;

  memset((char *)&clnt_res, 0, sizeof(clnt_res));

  if (clnt_call (clnt, SOMMA,
        (xdrproc_t) xdr_Operandi, (caddr_t) argp,
        (xdrproc_t) xdr_int, (caddr_t) &clnt_res,
        TIMEOUT) != RPC_SUCCESS) {
     return (NULL);
  }
  return (&clnt_res);
}
```

versione programma

FILE OPERAZIONI_CLNT.C: STUB DEL CLIENT 2/2




```
int * moltiplicazione_1(Operandi *argp, CLIENT *clnt)
{ static int clnt_res;

  memset((char *)&clnt_res, 0, sizeof(clnt_res));

  if (clnt_call (clnt, MOLTIPLICAZIONE,
        (xdrproc_t) xdr_Operandi, (caddr_t) argp,
        (xdrproc_t) xdr_int, (caddr_t) &clnt_res,
       TIMEOUT) != RPC_SUCCESS) {
     return (NULL);
  }
  return (&clnt_res);
}
```

FILE OPERAZIONI_SVC.C: STUB DEL SERVER 1/3

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

```
#include …

static void operazioniprog_1(struct svc_req *rqstp,
      register SVCXPRT *transp)
{  union { Operandi somma_1_arg; moltiplicazione_1_arg; } argument;
   char * result;
   xdrproc_t _xdr_argument, _xdr_result;
   char *(*local)(char *, struct svc_req *);

 switch (rqstp->rq_proc)
 { case NULLPROC:
      (void) svc_sendreply
                (transp, (xdrproc_t) xdr_void, (char *)NULL);
      return;
   case SOMMA:
      _xdr_argument = (xdrproc_t) xdr_Operandi;
      _xdr_result = (xdrproc_t) xdr_int;
      local = ( char *(*)(char *, struct svc_req *))somma_1_svc;
      break;
   case MOLTIPLICAZIONE:
      _xdr_argument = (xdrproc_t) xdr_Operandi;
      _xdr_result = (xdrproc_t) xdr_int;
      local = ( char *(*)(char *, struct svc_req *))moltiplicazione_1_svc;
      break;
   default:
      svcerr_noproc (transp);
      return;
   }
```

FILE OPERAZIONI_SVC.C: STUB DEL SERVER 2/3

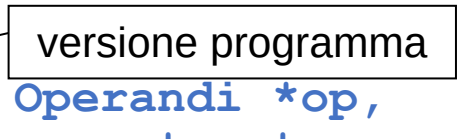
```
   memset ((char *)&argument, 0, sizeof (argument));

   if (!svc_getargs (transp, _xdr_argument, (caddr_t) &argument))
   { svcerr_decode (transp); return;}

   result = (*local)((char *)&argument, rqstp);
   if (result != NULL &&
       !svc_sendreply(transp, _xdr_result, result))
   { svcerr_systemerr (transp);}

   if (!svc_freeargs (transp, _xdr_argument, (caddr_t) &argument)){
      fprintf (stderr, "%s", "unable to free arguments");
      exit (1);
   }
   return;
 }
/* si è preparata la invocazione e si procede per passi:
  - prima la estrazione del parametro di ingresso
  - poi la invocazione
  - infine la estrazione del risutlato
  - e la liberazione dei parametri
*/
```

FILE OPERAZIONI_SVC.C: STUB DEL SERVER 3/3

```
int main (int argc, char **argv)
{  register SVCXPRT *transp;
   pmap_unset (OPERAZIONIPROG, OPERAZIONIVERS);
   /* doppio gestore di trasporto UDP e TCP*/

   transp = svcudp_create(RPC_ANYSOCK);
   if (transp == NULL)
   { fprintf (stderr, "%s","cannot create udp service.");exit(1);}
   if (!svc_register(transp, OPERAZIONIPROG, OPERAZIONIVERS,
                     operazioniprog_1, IPPROTO_UDP))
       { fprintf (stderr, "%s","unable to ..."); exit(1);}

   transp = svctcp_create(RPC_ANYSOCK, 0, 0);
   if (transp == NULL)
   {fprintf (stderr, "%s", "cannot create tcp service."); exit(1);}
   if (!svc_register(transp, OPERAZIONIPROG, OPERAZIONIVERS,
                     operazioniprog_1, IPPROTO_TCP))
   { fprintf (stderr, "%s", "unable to register ..."); exit(1);}

   svc_run ();

   /* NOTREACHED */
   fprintf (stderr, "%s", "svc_run returned");  exit (1);
}
```

FILE OPERAZIONI_PROC.C: IMPLEMENTAZIONE PROCEDURE 1/2

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

int *somma_1_svc( Operandi *op,
                  struct svc_req *rp)
{ static int ris;

  printf("Operandi ricevuti: %i e %i\n",
                                op->op1, op->op2);
  ris = (op->op1) + (op->op2);
  printf("Somma: %i\n", ris);
  return (&ris);
}
```



Notiamo **che si accede ai parametri tramite un puntatore che li racchiude tutti e ne permette l'accesso**
Il risultato è una variabile statica

FILE OPERAZIONI_PROC.C: IMPLEMENTAZIONE PROCEDURE 2/2

```
int *moltiplicazione_1_svc(Operandi *op, struct svc_req *rp)
{ static int ris;

  printf("Operandi ricevuti: %i e %i\n",
                                    op->op1, op->op2);
  ris = (op->op1) * (op->op2);
  printf("Moltiplicazione: %i\n", ris);
  return (&ris);
}
```

FILE OPERAZIONI_CLIENT.C: IMPLEMENTAZIONE DEL CLIENT 1/2

```
#include ...
main(int argc, char *argv[]){
   char *server; Operandi op;  CLIENT *cl; int *ris;

   if (argc != 5)   // controllo argomenti
   {fprintf(stderr, "uso:%s host tipo op1 op2\n", argv[0]);
      exit(1);}
   if (argv[2][0] != 'm' && argv[2][0] != 's')
   { fprintf(stderr, "uso: %s host somma/moltiplicazione op1 op2\n",
      argv[0]);
     fprintf(stderr, "tipo deve iniziare per 's' o 'm'\n");
     exit(1);}
   server = argv[1];
   op.op1 = atoi(argv[3]); op.op2 = atoi(argv[4]);

// creazione gestore di trasporto
   cl = clnt_create(server, OPERAZIONIPROG,
      OPERAZIONIVERS, "udp");
   if (cl == NULL) { clnt_pcreateerror(server);exit(1);}
```

FILE OPERAZIONI_CLIENT.C: IMPLEMENTAZIONE DEL CLIENT 2/2

```
   if(argv[2][0] == 's')
   ris = somma_1(&op, cl);
   if(argv[2][0] == 'm')
   ris = moltiplicazione_1(&op, cl);

   /* errore RPC */
   if (ris == NULL)
      { clnt_perror(cl, server); exit(1); }
   /* errore risultato: assumiamo che non si possa ottenere 0 */
   if (*ris == 0)
     { fprintf(stderr, "%s:...", argv[0], server); exit(1);}
   printf("Risultato da %s: %i\n",
                        server, *ris);

   // libero la risorsa gestore di trasporto
    clnt_destroy(cl);
}
```

COMPILAZIONE ED ESECUZIONE

Compilazione per generare l'eseguibile del **client**:

```
gcc -o operazioni  operazioni_client.c
        operazioni_clnt. operazioni_xdr.c
```

→ produce il comando **`operazioni`**

Compilazione per generare l'eseguibile del **server**:

```
gcc -o operazioni_server  operazioni_proc.c
        operazioni_svc.c operazioni_xdr.c
```

→ produce il comando **`operazioni_server`**

Esecuzione

1. Mandare in esecuzione il server con il comando: **`operazioni_server`**
2. Mandare in esecuzione il client con il comando:

   `operazioni serverhost somma op1 op2`

N.B.: nella variante il *client* viene lanciato con un unico argomento, *serverhost*, il *tipo di operazione* e gli *operandi* vengono recuperati interattivamente durante il ciclo d'esecuzione.

SPECIFICA: ESERCIZIO 2

Sviluppare un'applicazione C/S che consente di **ottenere l'echo di una stringa invocando una procedura remota**

Il **Client** è un filtro e realizza l'interazione con l'utente richiedendo una stringa, invoca la **procedura remota echo** passando come parametro **la stringa letta**, e stampa a video la **stringa ottenuta come valore di ritorno** dell'operazione invocata; si ripetono queste tre operazioni fino alla fine dell'interazione con l'utente

Il **Server** realizza il servizio di echo che **restituisce come risultato la stringa** passata come parametro di ingresso

Variante: realizzare il client non ciclico, che prende la stringa come argomento da linea di comando

FILE ECHO.X

```
program ECHOPROG {
  version ECHOVERS {
    string ECHO(string) = 1;
  } = 1;
} = 0x20000013;
```

Compilazione per generare il file header, il file per le conversioni xdr e gli stub:

```
rpcgen echo.x
```

Produce i file:

- **echo.h** da includere in **echo_proc.c** e in **echo_client.c**
- **echo_clnt.c** stub del client
- **echo_svc.c** stub del server

FILE ECHO_PROC.C: IMPLEMENTAZIONE PROCEDURE SERVER

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

char **echo_1_svc (char **msg, struct svc_req *rp)
{
  static char *echo_msg;

  free(echo_msg);
  echo_msg=(char*)malloc(strlen(*msg)+1);

  printf("Messaggio ricevuto: %s\n", *msg);
  strcpy(echo_msg, *msg);
  printf("Messaggio da rispedire: %s\n", echo_msg);

  return (&echo_msg);
}
```

Motivi della posizione della ***"free"***:
Prima della malloc perché serve a liberare la memoria occupata dall'invocazione precedente

→ ***inutile*** per la ***prima volta***, ***ok le successive***

FILE ECHO_CLIENT.C: IMPLEMENTAZIONE DEL CLIENT 1/2

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "echo.h"
#define DIM 100

main(int argc, char *argv[]){
  CLIENT *cl;
  char **echo_msg;
  char *server;
  char *msg;

  if (argc < 2){fprintf(stderr, "uso: ...", argv[0]);
  exit(1);}
  server = argv[1];
  cl = clnt_create(server, ECHOPROG, ECHOVERS, "udp");
  if (cl == NULL)
    { clnt_pcreateerror(server);exit(1);}

  msg=(char*)malloc(DIM);
  printf("Qualsiasi tasto per procedere, EOF per
  terminare: ");      printf("Messaggio (max 100
  caratteri)? ");
   /* lettura della stringa da inviare o fine file*/
```

FILE ECHO_CLIENT.C: IMPLEMENTAZIONE DEL CLIENT 2/2

```
while (gets(msg))
  {
    echo_msg = echo_1(&msg, cl);

    if (echo_msg == NULL)  /* controllo errore RPC */
   { fprintf(stderr, "%s: %s fallisce la rpc\n", argv[0],server);
    clnt_perror(cl, server);  exit(1);
   }
    if (*echo_msg == NULL)  /* controllo errore risultato */
   { fprintf(stderr, "%s: ...",   argv[0], server);
     clnt_perror(cl, server);  exit(1);
   }
    printf("Messaggio consegnato a %s: %s\n", server, msg);
    printf("Messaggio ricevuto da %s: %s\n", server, *echo_msg);

    printf("Qualsiasi tasto per procedere, EOF per terminare: ")
    printf("Messaggio (max 100 caratteri)? ");
     /* lettura della stringa da inviare o fine file*/
  } // while gets(msg)

  free(msg); clnt_destroy(cl);
  // Libero risorse: malloc e gestore di trasporto
  printf("Termino...\n");  exit(0);
} // main
```

ALLOCAZIONE MEMORIA LATO SERVER E CLIENT

Lato server il parametro di uscita deve essere **static**, inoltre:

- È necessario **allocare esplicitamente memoria** per il parametro di uscita (es., **`echo_msg`** in **`echo_proc.c`**)
- **Non** serve **sul parametro di ingresso**: l'allocazione è fatta automaticamente dal supporto rpc (es., **`msg`** in **`echo_proc.c`**)

Lato client:

- È necessario **allocare esplicitamente memoria** per il parametro di ingresso (es., **`msg`** in **`echo_client.c`**)
- **Non** serve **sul parametro di uscita**: l'allocazione del valore di ritorno è fatta automaticamente dal supporto rpc (es., **`echo_msg`** in **`echo_client.c`**)

COMPILAZIONE ED ESECUZIONE

Compilazione per generare l'eseguibile del **client**:

```
gcc echo_client.c echo_clnt.c -o remote_echo
```

→ produce il comando `remote_echo`

Compilazione per generare l'eseguibile del **server**:

```
gcc echo_proc.c echo_svc.c -o echo_server
```

→ produce il comando `echo_server`

Esecuzione

1. Mandare in esecuzione il server con il comando: `echo_server`
2. Mandare in esecuzione il client con il comando: `remote_echo serverhost`

N.B.: nella variante il *client* viene lanciato con due argomenti: *serverhost* e la *stringa* che si vuole inviare al server

ALCUNE OPZIONI DI RPCGEN (VEDERE IL MAN):

-a Generate all the files including sample code for client and server side

-Sc Generate sample code to show the use of remote procedure and how to bind to the server before calling the client side stubs generated by rpcgen

-Ss Generate skeleton code for the remote procedures on the server side. You would need to fill in the actual code for the remote procedures

Verificare le altre opzioni

REPETITA: XDR - ALCUNI CONSIGLI SULLA DEFINIZIONE DEI TIPI DI DATI

I dati al **primo livello** (cioè quelli passati direttamente alle funzioni) possono essere passati **SOLO per valore** e **NON si possono passare** tipi di dato complessi (ad esempio gli array). Ad esempio:

```
string ECHO(string s);
```
Sì ☺

```
char[] ECHO(char arrayCaratteri[12]);
```
No ☹

I dati al **secondo livello** (cioè definiti all'interno di altre strutture dati) possono invece usare anche strutture dati complesse (ad esempio array) e puntatori.

```
struct Input{char arrayCaratteri[12];};
… Input ECHO(Input i);
```
Sì ☺

Le **matrici** vanno però sempre definite **PER PASSI**:

```
struct Matrix{char arrayCaratteri[12][12];};
```
No ☹

```
struct Riga{char arrayCaratteri[12];};
struct Matrix{Riga riga[12];};
```
Sì ☺