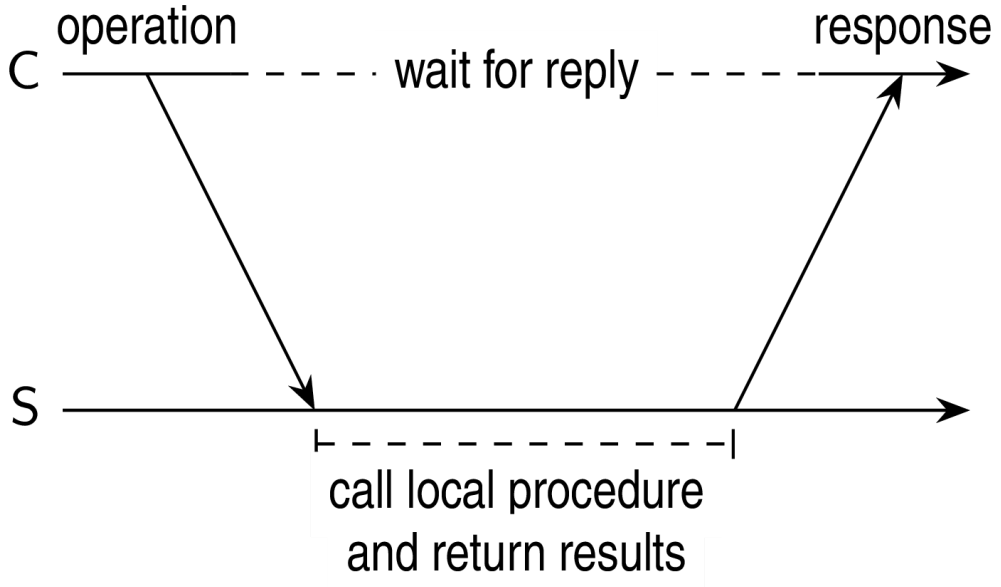
Programação C/S Usando RPC – Chamada de Procedimento Remoto

Sistemas distribuídos em geral são baseados na troca de mensagens entre processos. Dentre os mecanismos de troca disponíveis, as Chamadas de Procedimento Remoto ou RPC (Remote Procedure Call´s) são consideradas até o momento como um pilar básico para a implementação de boa parte dos requisitos de Sistemas Distribuídos (escalabilidade, transparência etc.). Por esse motivo, faz-se necessário um estudo mais aprofundado sobre o método de programação usando RPC. De um modo geral, pode-se dizer que as chamadas de procedimento remoto são idênticas às chamadas de procedimento local, com a exceção de que as funções chamadas ficam residentes em hosts distintos. Nesse contexto, um host executando o programa principal ou cliente aciona uma chamada de procedimento remoto (idêntico ao método de programação estruturada convencional) e ficaria aguardando um resultado. Por outro lado, um outro host, denominado servidor teria as referidas funções em execução no seu espaço de memória e ficaria aguardando requisições para as mesmas. Ao chegar uma requisição, o servidor executa a função identificada e retorna os resultados para o cliente, conforme demonstra a Figura a seguir:



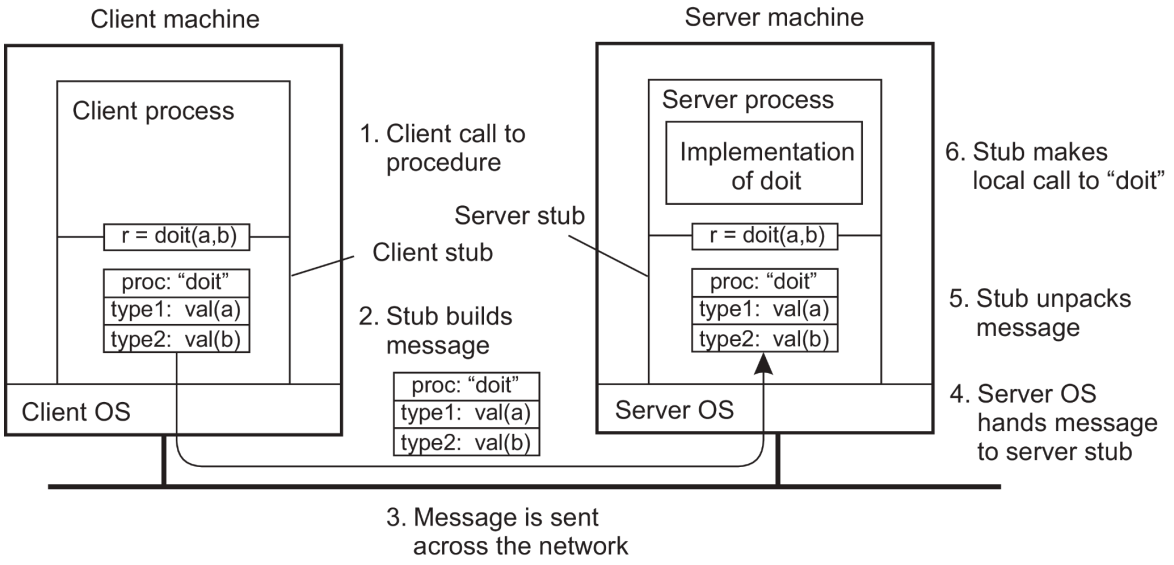
Diante do que foi dito, pode o leito fazer alguns questionamentos tais como:

- Como o cliente consegue passar para um outro host seus parâmetros de chamada?
- Como o servidor consegue individualizar cada função e saber qual delas está sendo acionada?
- Que mecanismos os lados cliente e servidor devem possuir para viabilizar chamadas remotas?

Para responder essas e outras perguntas a respeito da comunicação RPC, esse documento apresenta uma relação de tópicos que inclui uma parte teórica (itens 1 a 4) e um exemplo de conversão de um programa convencional em um programa RPC (item 5).

1. Princípio da comunicação RPC entre um programa cliente e um Servidor

O objetivo da biblioteca RPC é permitir ao programador uma forma de escrever o seu código de forma similar ao método adotado para a programação convencional. Para isso, a estrutura RPC define um esquema de encapsulamento de todas as funções associadas à conexão remota num pedaço de código chamado de STUB. Dessa maneira, o código do usuário terá um comportamento similar ao exemplo que está apresentado na Figura, a seguir:



Perceba que, da forma como está apresentado, existirá um STUB associado ao código do cliente e outro associado ao
código do servidor. Dessa forma, o diálogo dos módulos cliente e servidor acontecerá com ajuda dos STUB´s, de acordo
com a seguinte sequência:

A. O cliente chama uma função que está implementada no stub do cliente
B. O stub do cliente constrói uma mensagem e chama o Sistema Operacional local
C. O sistema operacional do cliente envia a mensagem pela rede para o sistema operacional do host remoto
D. O sistema operacional remoto entrega a mensagem recebida para o stub instalado no servidor
E. O stub servidor desempacota os parâmetros e chama a aplicação servidora, mais especificamente, a função associada à função que estava no stub do cliente
F. O servidor faz o trabalho que deve fazer e retorna o resultado para o stub do servidor
G. O stub do servidor empacota os dados numa mensagem e chama o sistema operacional local
H. O sistema operacional do servidor envia a mensagem para o sistema operacional do cliente
I. O sistema operacional do cliente entrega a mensagem recebida para o stub do cliente
J. O stub desempacota os resultados e retorna-os para o cliente

Apesar de todos esses passos e mecanismos de encapsulamento, pricipalmente considerando as funções de rede, a programação RPC é viável pelos seguintes motivos:

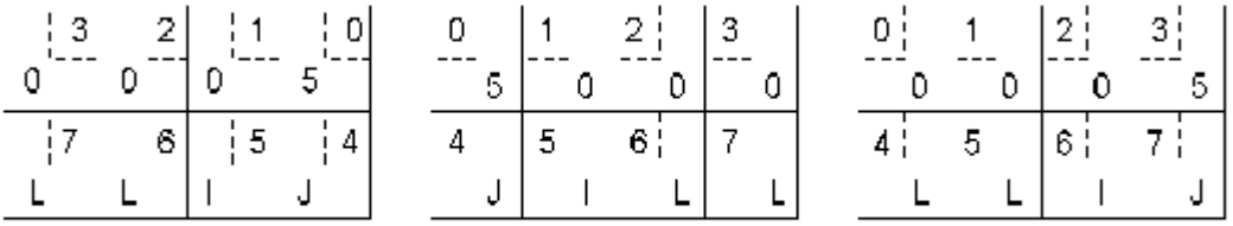
- A complexidade de encapsulamento e geração de stubs, arquivos de include e todas as chamadas de rede não são responsabilidade do programador. Em outras palavras, a biblioteca RPC contém ferramentas que auxiliam na geração dos módulos mencionados
- Caberá ao programador alterar apenas os arquivos relacionados ao “cliente” e ao “servidor”, inserindo nesses arquivos a lógica desejada. Essas alterações não incluem nenhum aspecto referente aos serviços de rede ou de como localizar o servidor.

Portanto, considerando o encapsulamento das funções de rede nos stubs cliente e servidor, a programação RPC se aproxima da programação convencional[1]. Essa noção vai ficar mais clara à medida que os itens seguintes forem sendo compreendidos.

2 Conversão de tipos de dados entre os módulos de programa Cliente e Servidor

Uma das preocupações das implementações da biblioteca RPC é a necessidade de prover comunicação entre sistemas abertos. Ou seja, a capacidade de fazer com que funções

possam ser escritas entre hosts que possuem diferentes representações internas para números (inteiro, real, etc.) e caracteres seja viabilizado. Para exemplificar, a Figura abaixo apresenta uma situação de diálogo onde um cliente instalado num microcomputador Pentium quer passar a sequência LLIJ para um servidor instalado numa SPARC Station. Conforme pode ser visto, a representação da sequência LLIJ numa máquina pentium (a) está invertida em relação ao que está representado internamente na SPARC Station após ter sido recebido (b). Nesse caso será necessário algum mecanismo de inversão desses caracteres para que o receptor compreenda o que emissor quis comunicar (a letra (c) apresenta esse modificação). No caso da figura, os números menores em cada quadro indicam o endereço de cada byte.



(a) (b) (c)

Para resolver essa questão de representação de dados entre hosts distintos existe uma biblioteca associada à biblioteca RPC denominada XDR (eXternal Data Representation), cuja funcionalidade é apresentar uma padronização entre tipos de dados de máquinas heterogêneas. Para isso, a biblioteca define uma série de tipos de dados padronizados, os quais estão apresentados na Tabela a seguir:

Tipo de Dado	**Tam.**	**Descrição**
Int	32	Inteiro binário sinalizado de 32 bits
Unsigned int	32	Inteiro binário não sinalizado de 32 bits
Bool	32	Valor booleano (false ou true), representado por 0 ou 1
Enum	Arbitrário	Tipo de enumeração com valores definidos por inteiros (ex.:red=1, blue=2)
Hyper	64	Inteiro sinalizado de 64 bits
Unsigned hyper	64	Inteiro sinalizado de 64 bits
Float	32	Numero de ponto flutuante com precisão simples
Double	64	Número de ponto flutuante com precisão dupla
Opaque	Arbitrário	Dado não convertido, i.e, dado na representação nativa do remetente
String	Arbitrário	String de caracteres
Fixed Array	Arbitrário	Um array de tamanho fixo de qualquer outro tipo de dado
Counted Array	Arbitrário	Um array no qual o tipo tem um limite superior fixo, mas arrays individuais podem variar no tamanho
Structure	Arbitrário	Um dado agregado, como uma struct da linguagem C
Discriminated Unidon	Arbitrário	Uma estrutura de dados que permite uma de várias formas alternativas, como uma union da linguagem C
Void	0	Usado se nenhum dado está presente onde um item de dado é opcional (ex.: dentro de uma structure)
Symbolic	Arbitrário	Uma constante simbólica e um valor associado

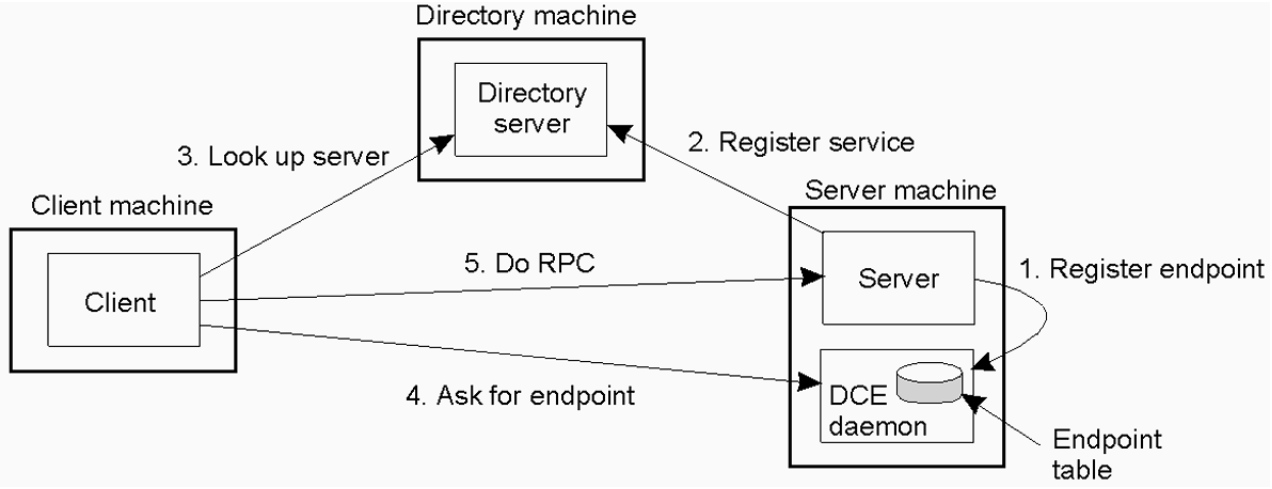
Contant		
Optional Data	Arbitrário	Permite zero ou uma ocorrência de um item

Além desses tipos de dados, a biblioteca XDR precisa ter um conjunto de funções que permitem realizar as conversões entre os tipos de dados locais para o formato XDR. Dentre as funções existentes, algumas das mais usuais estão apresentadas na Tabela a seguir:

Função	**Argumentos**	**Tipo de dado convertido**
Xdr_bool	Xdrs, ptrbool	Booleano (int em C)
Xdr_bytes	Xdrs, ptrstr, strsize, maxsize	Byte String contado
Xdr_char	Xdrs, ptrchar	Caracter
Xdr_double	Xdrs, ptrdouble	Ponto flutuante de precisão dupla
Xdr_enum	Xdrs, ptrint	Variável de tipo de dado enumerado (um int em C)
Xdr_float	Xdrs, ptrfloat	Ponto flutuante de precisão simples
Xdr_int	Xdrs, ip	Inteiro de 32 bits
Xdr_long	Xdrs, ptrlong	Inteiro de 64 bits
Xdr_opaque	Xdrsptrchar, count	Bytes enviados sem uma conversão
Xdr_pointer	Xdrs, ptrobj, objsize, xdrobj	Um ponteiro (usado em listas encadeadas ou árvores)
Xdr_short	Xdrs	Inteiro de 16 bits
Xdr_string	Xdrs, ptrstr, maxsize	Uma string em linguagem C
Xdr_u_char	Xdrs, ptruchar	Inteiro não sinalizado de 8 bits
Xdr_u_int	Xdrs, ptrint	Inteiro não sinalizado de 32 bits
Xdr_u_long	Xdrs, ptrulong	Inteiro não sinalizado de 64 bits
Xdr_u_short	Xdrs, ptrushort	Inteiro não sinalizado de 16 bits
Xdr_union	Xdrsptrdiscrim, ptrunion, choicefcn, default	União discriminada
Xdr_vector	Xdrs, ptrarray, size, elemsize, elemproc	Array de tamanho fixo
Xdr_void	-	Nõo é uma conversão (uso: denotar a parte vazia de uma estrutura de dados)

3. Como o Cliente Localiza o Servidor RPC

Toda aplicação RPC cliente servidora em geral é baseada num serviço de diretórios. Em outras palavras, o cliente, para localizar o servidor RPC remoto precisa questionar algum processo anterior que o informe sobre a porta onde determinado serviço está escutando. Na Figura a seguir está um exemplo dos passos relacionados ao processo de localização do servidor:



O serviço de diretórios então é um pré-requisito para viabilizar aplicações RPC. Em ambiente Linux, por exemplo, a implementação desse serviço de diretórios relacionados aos servidores RPC denomina-se portmapper. Dessa forma podese concluir que no Linux, para executar qualquer servidor RPC será preciso antes executar o ***rpcbind***.

Falando mais especificamente sobre o RPC portmapper, este é um servidor que converte números de programa RPC em números de portas disponíveis no protocolo TCP/IP. De acordo com a Figura acima, quando um servidor RPC é executado uma das primeiras providências que ele faz é "dizer" ao serviço de diretórios (nesse caso, o portmapper) qual número de porta ele está escutando e que números de programa ele está preparado para servir. Quando um cliente deseja fazer uma chamada RPC para um dado número de programa, ele irá primeiro contactar o portmapper na máquina servidora para determinar o número da porta onde os pacotes RPC devem ser enviados. Um servidor RPC provê um grupo de procedures ou funções as quais o cliente pode invocar enviando uma requisição para o servidor. O servidor então invoca a procedure mencionada pelo cliente, retornando um valor quando necessário. A coleção de procedures que um servidor RPC provê é chamada de um programa e é identificado por um número de programa. No Linux, o arquivo /etc/rpc mapeia nomes de serviço para seus números de programa. Um exemplo do arquivo /etc/rpc é apresentado abaixo:

```
# This file contains user readable names that can be used in place of rpc
# program numbers.

portmapper      100000  portmap sunrpc rpcbind
rstatd          100001  rstat rstat_svc rup perfmeter
rusersd         100002  rusers
nfs             100003  nfsprog
ypserv          100004  ypprog
mountd          100005  mount showmount
ypbind          100007
walld           100008  rwall shutdown
```

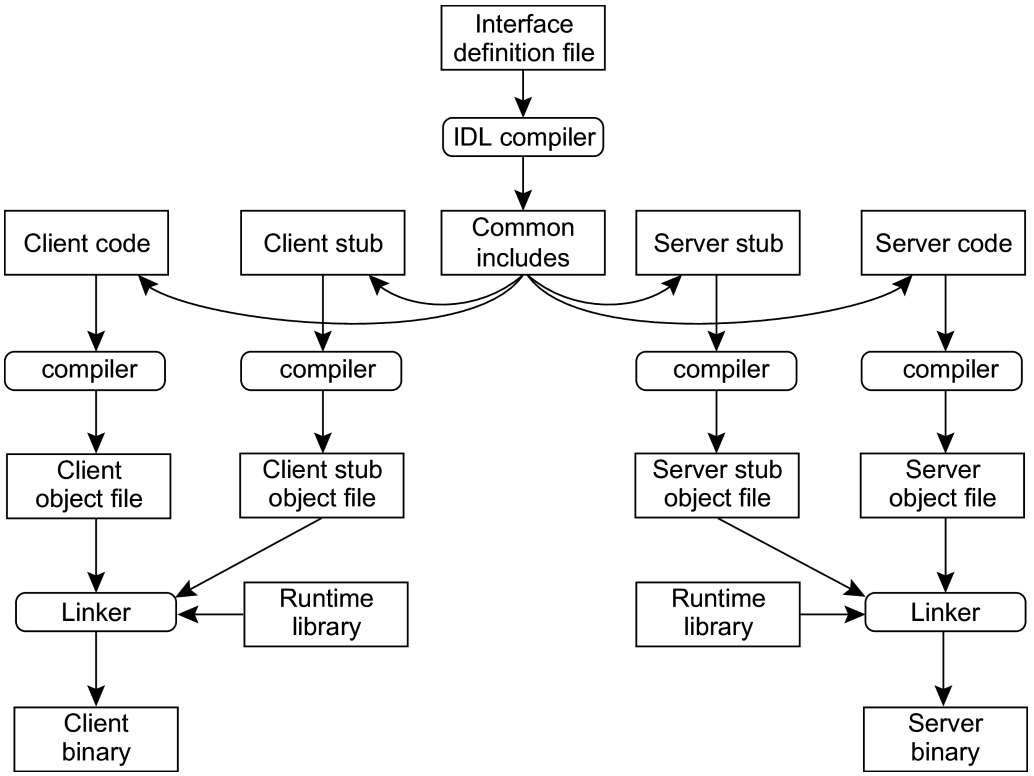
Portanto, esse arquivo só deve ser alterado se um novo serviço RPC for introduzido no servidor.

Para instalar o ***rpcbind***, utilize os seguintes comando no linux:

```
sudo apt-get install rpcbind
sudo nano /etc/default/rpcbind
```

4. Usando o rpcgen para gerar os stubs do Cliente e do Servidor

Conforme já foi mencionado, junto com a biblioteca RPC existe uma ferramenta denominada rpcgen cujo objetivo é gerar, a partir de um arquivo de definição de interface (IDF – Interface Definition File), todos os módulos de software que devem estar nos lados cliente e servidor, incluindo os stubs, conforme demonstra a figura abaixo:



De um modo geral os passos para geração de uma aplicação cliente/servidor RPC os seguintes passos devem ser considerados:

A. Identificar quais funcionalidades devem estar no programa principal e quais sub-rotinas serão acionadas no servidor. Essa percepção deve ser sistematizada no arquivo de definição de interface (Interface Definition File) apresentando na figura acima. Em outras palavras, o programador deve construir seu código usando linguagem C convencional e, a partir desse código, identificar que funções devem ser ativadas e que parâmetros devem ser passados para elas. Esses informações devem ser incluídas no arquivo de interface IDF e, a partir dele, gera-se todos os códigos da figura acima.

B. Aplicar o utilitário de geração dos módulos cliente e servidor no arquivo IDF gerado. No caso do Linux, a ferramenta de geração dos módulos da figura é o ***rpcgen*** da SUN Microsystens, considerada um padrão de facto no mercado. Esse utilitário pressupõe que o arquivo IDF tem um nome com a sufixo **.x**, e com base nele, os códigos são gerados em linguagem C e estão estruturados de forma que o programador possa alterá-los com o menor esforço possível.

C. Modificar os arquivos do cliente e do servidor para que atendam o objetivo desejado para a aplicação. Em princípio o programador necessita mexer apenas nesse dois arquivos, inserindo neles as lógicas presentes nas funções principal e secundárias do código que foi construído no modo convencional.

D. Compilar os códigos alterados e colocá-los em hosts cliente e servidor. No caso do rpcgen da SUN instalado em máquinas Linux, o rpcgen gera, além dos arquivos mencionados, um arquivo com diretivas de compilação para ajudar no processo de geração dos binários cliente e servidor. Esse arquivo é um **Makefile.progr** que deverá ser utilizado pelo utilitário make.

Com relação às modificações que um programa RPC pode ter, alguns cuidados podem ser tomados:

- Se houver alteração na lógica dos módulos cliente e/ou servidor que não comprometa a passagem de parâmetros, o correto é apenas modificar os módulos e executar novamente o comando make
- Se as alterações desejadas influenciarem no tipo de dados dos parâmetros ou na definição das funções que estão no servidor, então será necessário regerar os códigos cliente e servidor com novo uso da ferramenta rpcgen.
- Sobre o arquivo IDF, uma boa estratégia na passagem de parâmetros é encapsulá-los em uma estrutura de dados cujos campos é cada um dos parâmetros das funções. Dessa forma o cliente passa para o servidor um único parâmetro que é uma estrutura de dados. Por outro lado, o receptor deverá acessar nessa estrutura de dados recebida, o campo que lhe interessa para realizar suas tarefas (vide seção 5 para mais clareza).
- O processo ***rpcbind*** deve estar sempre ativo no equipamento servidor
- O lado servidor deve sempre ser acionado primeiro e o cliente sempre deve ter como parâmetro de entrada o nome ou ip de localização do servidor.

5. Exemplo de construção de um program RPC a partir de um programa convencional

Vamos supor que desejamos construir uma aplicação distribuída cujo objetivo é receber, via teclado, dois números inteiros e
retornar o resultado da soma e subtração entre esses números. No caso da aplicação distribuída RPC o cliente se encarregará de receber os parâmetros e passá-los ao servidor. Caberá ao servidor executar os cálculos e retornar os resultados para que o cliente possa imprimir na tela do cliente. Para alcançar esse objetivo, os seguintes passos são necessários:

a) Gerar a aplicação de forma modular usando programação convencional

No caso de uma aplicação convencional, esse código poderia ser dividido em programa principal e duas funções add e sub, conforme demonstra o código abaixo:

```c
#include <stdio.h>

int add(int x, int y)
{
    int result;
    printf("Requisicao de adicao para %d e %d\n", x, y);
    result = x + y;
    return (result);
} /* fim funcao add */

int sub(int x, int y)
{
    int result;

    printf("Requisicao de subtracao para %d e %d\n", x, y);
    result = x - y;
    return (result);
} /* fim funcao sub *

int main(int argc, char *argv[])
```

```
{
    int x, y;

    if (argc != 3)
    {
        fprintf(stderr, "Uso correto: %s num1 num2\n", argv[0]);
        exit(0);
    }

    /* Recupera os 2 operandos passados como argumento */
    x = atoi(argv[1]);
    y = atoi(argv[2]);
    printf("%d + %d = %d\n", x, y, add(x, y));
    printf("%d - %d = %d\n", x, y, sub(x, y));
    return (0);

} /* fim main */
```

No caso do exemplo apresentado, está claro que existem duas funções ADD e SUB a serem implementadas e as duas variáveis X e Y devem ser passadas como parâmetros entre programa principal e as funções remotas.

b) Gerar o arquivo de definição de Interface (IDF) a partir do código apresentado no item (a):

A idéia é fazer com que o programa principal seja executado em uma máquina e as funções add e sub sejam executadas em uma outra máquina. Para isso, a maneira mais eficiente é lançar mão da ferramenta rpcgen que vem junto com a biblioteca padrão rpc distribuída pela SUN (padrão de facto). Essa ferramenta consegue gerar todos os códigos mencionados a partir de um arquivo IDF que contém basicamente:

- A definição dos parâmetros que vão ser passados para a(s) procedure(s) remota(s)
- A definição das funções remotas

Seguindo essa visão, pode-se então gerar um IDF cujo nome será, nesse exemplo, calcula.x e terá o conteúdo especificado o código abaixo (***arquivo calcula.x***):

```
struct operandos {
    int x;
    int y;
};

program PROG {
   version VERSAO {
     int ADD(operandos) = 1;
     int SUB(operandos) = 2;
   } = 100;
} = 55555555;
```

Nesse código , pode-se observar o seguinte:

- a struct operandos é a estrutura de dados contendo os inteiros x e y que vai ser passada para as funções remotas. Essa definição segue a recomendação de

simplicidade de programação na qual se especifica que a passagem deuma única variável (mesmo que composta) é melhor do que passar muitas variáveis individuais.

- a definição program **PROG** define o número do programa **RPC** como **55555555**. Ou seja, o programa terá um número de identificação que deverá ser reconhecido tanto no cliente quanto no servidor. Além disso, é importante perceber que essa definição trás um número de versão (**no caso 100**) para esse programa e também um código para cada uma das duas rotinas declaradas. Dessa forma, a função **ADD** terá o código **1** e a função **SUB** terá o código **2**. Vale observar que essa notação não é feita em linguagem C, apesar da similaridade percebida.

Uma vez gerado esse arquivo, pode-se aplicar a ferramenta rpcgen para que os arquivos-fonte em linguagem C dos lados cliente e servidor sejam gerados. O ***rpcgen*** possui vários parâmetros de ativação mas, no caso desse exemplo, vamos utilizar o seguinte:

```
rpcgen -a calcula.x
```

Os arquivos gerados serão os seguintes:

ARQUIVO	**CONTEÚDO**
calcula.h	Arquivo com as definições que deverão estar inclusas nos códigos cliente e servidor
calcula_client.c	Arquivo contendo o esqueleto do programa principal do lado cliente
calcula_clnt.c	Arquivo contendo o stub do cliente
Calcula_xdr.c	Contém as funções xdr necessárias para a conversão dos parâmetros a serem passados entre hosts
Calcula_svc.c	Contém o programa principal do lado servidor
Calcula_server.c	Contém o esqueleto das rotinas a serem chamadas no lado servidor
Makefile.calcula	Deve ser renomeado para Makefile. Contém as diretivas de compilação para a ferramenta **make**.

Algumas observações importantes sobre cada um desses arquivos:

calcula.h – Esse arquivo contém, dentre outras definições, o relacionamento entre os nomes (de funções, de programa, versão etc) e os seus respectivos códigos, bem como a declaração da estrutura de dados que deve ser passada como parâmetro entre cliente e servidor. Todas as definições desse arquivo foram provenientes das declarações feitas no arquivo calcula.x.

calcula_client.c – Nesse arquivo pode-se destacar as seguintes características:
A tabela a seguir contém o código exemplo de calcula_client.c:

```c
#include "calcula.h"

void
prog_100(char *host)
{
    CLIENT *clnt;
    int  *result_1;
    operandos  add_100_arg;
    int  *result_2;
```

```
    operandos  sub_100_arg;

#ifndef DEBUG
    clnt = clnt_create (host, PROG, VERSAO, "udp");
    if (clnt == NULL) {
        clnt_pcreateerror (host);
        exit (1);
    }
#endif  /* DEBUG */

    result_1 = add_100(&add_100_arg, clnt);
    if (result_1 == (int *) NULL) {
        clnt_perror (clnt, "call failed");
    }
    result_2 = sub_100(&sub_100_arg, clnt);
    if (result_2 == (int *) NULL) {
        clnt_perror (clnt, "call failed");
    }
#ifndef DEBUG
    clnt_destroy (clnt);
#endif   /* DEBUG */
}

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

    if (argc < 2) {
        printf ("usage: %s server_host\n", argv[0]);
        exit (1);
    }
    host = argv[1];
    prog_100 (host);
    exit (0);
}
```

Este programa contém duas funções: (i) o programa principal (main), que recebe o nome do host remoto via parâmetro de entrada e, (ii) a função prog_100 que recebe o nome do host remoto. Essa última realiza o seguinte:

- chama a função clnt_create cujo objetivo é contactar gerar uma conexão com o portmapper instalado no host remoto e questionando a ele qual é o número da porta do servidor RPC
- chama as duas funções add e sub presentes no stub local, passando para as mesmas as variáveis add_100_arg e sub_100_arg (do tipo operandos). É importante perceber que essas variáveis não estão preenchidas nesse código. Portanto, uma das alterações no cliente é preencher essas variáveis com os dados que devem ser passados às funções remotas.

As funções chamadas no stub cliente estão descritas a seguir.

calcula_clnt.c – Contém as funções referenciadas no código calcula_client.c, ou seja, as funções add_100 e sub_100, conforme demonstra o desenho a seguir:

```c
#include <memory.h> /* for memset */
#include "calcula.h"

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

int *
add_100(operandos *argp, CLIENT *clnt)
{
    static int clnt_res;

    memset((char *)&clnt_res, 0, sizeof(clnt_res));
    if (clnt_call (clnt, ADD,
        (xdrproc_t) xdr_operandos, (caddr_t) argp,
        (xdrproc_t) xdr_int, (caddr_t) &clnt_res,
        TIMEOUT) != RPC_SUCCESS) {
        return (NULL);
    }
    return (&clnt_res);
}

int *
sub_100(operandos *argp, CLIENT *clnt)
{
    static int clnt_res;

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

Essas funções realizam os seguintes passos:

- Fazem o encapsulamento das chamadas do xdr através da chamada à função xdr_operandos, que por sua vez, está presente no arquivo calcula_xdr.c. Essa chamada é feita para que haja o encapsulamento das variáveis embutidas na estrutura

operandos no formado xdr e, como pode ser visto, está embutida na chamada à system call clnt_call, descrita a seguir.

- Fazem chamada à função clnt_call, que é responsável por enviar os argumentos no formato XDR para o servidor RPC remoto. Por esse motivo, a função clnt_call contém como parâmetros:
  - a identificação do servidor remoto (variável clnt),
  - identificação da função remota a ser chamada nesse servidor (variável ADD ou SUB),
  - parâmetros convertidos no formato XDR e,
  - uma variável para conter o resultado ofertado pela função remota chamada.

calcula_xdr.c – Contém as funções xdr a serem usadas pelo código, conforme apresentado anteriormente nas tabelas de tipos e funções nesse documento.

```c
#include "calcula.h"

bool_t
xdr_operandos (XDR *xdrs, operandos *objp)
{
    register int32_t *buf;

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

Nesse caso a função chamada é a ***xdr_int*** que converte os inteiros **x** e **y**, campos da estrutura de dados operandos, no formato XDR.

calcula_svc.c – É o arquivo que representa o stub do servidor. Este arquivo possui duas rotinas principais: (i) o programa principal que é responsável pelos controles iniciais de registro do servidor e, (ii) a função secundária prog_100 cujo objetivo é receber a identificação da função chamada e fazer o desvio para uma função local de acordo com esse identificação.

```c
#include "calcula.h"
#include <stdio.h>
#include <stdlib.h>
#include <rpc/pmap_clnt.h>
#include <string.h>
#include <memory.h>
#include <sys/socket.h>
#include <netinet/in.h>

#ifndef SIG_PF
#define SIG_PF void(*)(int)
#endif

static void
```

```
prog_100(struct svc_req *rqstp, register SVCXPRT *transp)
{
    union {
        operandos add_100_arg;
        operandos sub_100_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 ADD:
        _xdr_argument = (xdrproc_t) xdr_operandos;
        _xdr_result = (xdrproc_t) xdr_int;
        local = (char *(*)(char *, struct svc_req *)) add_100_svc;
        break;

    case SUB:
        _xdr_argument = (xdrproc_t) xdr_operandos;
        _xdr_result = (xdrproc_t) xdr_int;
        local = (char *(*)(char *, struct svc_req *)) sub_100_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)) {
        fprintf (stderr, "%s", "unable to free arguments");
        exit (1);
    }
    return;
}

int
```

```c
main (int argc, char **argv)
{
    register SVCXPRT *transp;

    pmap_unset (PROG, VERSAO);

    transp = svcudp_create(RPC_ANYSOCK);
    if (transp == NULL) {
        fprintf (stderr, "%s", "cannot create udp service.");
        exit(1);
    }
    if (!svc_register(transp, PROG, VERSAO, prog_100, IPPROTO_UDP)) {
        fprintf (stderr, "%s", "unable to register (PROG, VERSAO, udp).");
        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, PROG, VERSAO, prog_100, IPPROTO_TCP)) {
        fprintf (stderr, "%s", "unable to register (PROG, VERSAO, tcp).");
        exit(1);
    }

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

Dentre outras funcionalidades, o programa principal faz o seguinte:

- Verifica se esse programa servidor já não está instalado no portmapper (função pmap_unset). Se estiver solicita a remoção desse registro na tabela do portmapper
- Cria sockets **UDP** e **TCP** associando-os à portas livres no servidor (***svcudp_create*** e ***svctcp_create***)
- Registra essas portas no portmapper (função svc_register) vinculadas ao número do programa, da versão e a respectiva função que deve ser chamada (no caso prog_100) quando algum cliente desejar se conectar a esse servidor. Em outras palavras o número de programa e versão se referem à função secundária prog_100 residente nesse arquivo.
- Chama a função svc_run para habilitar a escuta permanente da porta que foi associada a esse servidor. Quando uma chamada acontece o svc_run promove o desvio para a função prog_100 definida no registro desse servidor. Implica dizer que o svc_run executa um loop infinito internamente aguardando por conexões remotas. Nesse caso a desativação do servidor deve ser abrupta através de um **<ctrl-c>**.

A função prog_100, por sua vez realiza as seguintes funções:

- Recebe os parâmetros de entrada, dentre eles a identificação da função (ADD ou SUB) chamada.
- Chama a função xdr_operandos para realizar o processo inverso de conversão, ou seja, do formato XDR para o formato local desse servidor
- Chama uma das funções (add_100_svc ou sub_100_svc) presentes no arquivo calcula_server.c
- Recebe o retorno da função chamada e converte esses valores novamente no formato xdr, através da chamada de funções dessa biblioteca
- Chama a função sendreply que retorna para o stub do cliente os resultados calculados pelas funções add_100_svc ou sub_100_svc.

calcula_server.c – Contém os códigos das funções que devem ser alterados para realizar o que se deseja.

```c
#include "calcula.h"

int *
add_100_svc(operandos *argp, struct svc_req *rqstp)
{
    static int  result;

    /*
     * insert server code here
     */

    return &result;
}

int *
sub_100_svc(operandos *argp, struct svc_req *rqstp)
{
    static int  result;

    /*
     * insert server code here
     */

    return &result;
}
```

Nesse caso, conforme pode ser visto, as funções ***add_100_svc*** e ***sub_100_svc*** possuem apenas o esqueleto das funções e devem ser alteradas para incluir a lógica desejada pelo programador, assim como é feito no arquivo **calcula_client.c**. Um exemplo de modificação possível no arquivo calcula_client.c está descrito no quadro a seguir:

```c
#include <stdio.h>
#include <unistd.h>
#include <arpa/inet.h>
```

```c
#include "add.h" /* Criado pelo rpcgen  */

int add(CLIENT *clnt, int x, int y)
{
    operandos ops;
    int *result;
    ops.x = x;
    ops.y = y;
    /* Chama o stub cliente criado pelo rpcgen */
    result = add_100(&ops, clnt);

    if (result == NULL)
    {
        fprintf(stderr, "Problema na chamada RPC\n");
        exit(0);
    }

    return (*result);

} /* fim função add local */

int sub(CLIENT *clnt, int x, int y)
{
    operandos ops;
    int *result;

    /* Preenche struct operandos p ser enviada ao outro lado */
    ops.x = x;
    ops.y = y;

    /* Chama o stub cliente criado pelo rpcgen */
    result = sub_100(&ops, clnt);

    if (result == NULL)
    {

        fprintf(stderr, "Problema na chamada RPC\n");

        exit(0);
    }

    return (*result);

} /* fim funcao sub local */

int main(int argc, char *argv[])
{
    CLIENT *clnt;
```

```
    int x, y;
    struct sockaddr_in server_addr;
    int socket_fd;
    int port = 111;

    if (argc != 4)
    {
        fprintf(stderr, "Uso: %s hostname num1 num2\n", argv[0]);

        exit(0);
    }

    // Set up the server address
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(port);

    if (inet_pton(AF_INET, argv[1], &server_addr.sin_addr) <= 0) {
        perror("Invalid server address");
        exit(1);
    }

    clnt = clnt_create(argv[1], PROG, VERSAO, "tcp");
    //clnt = clnttcp_create(&server_addr, PROG, VERSAO, &socket_fd, 0, 0);
    /* Garantindo a criacao da ligacao com o remoto */

    if (clnt == (CLIENT *)NULL)
    {
        clnt_pcreateerror(argv[1]);
        exit(1);
    }
    /* Recupera os 2 operandos passados como argumento */
    x = atoi(argv[2]);
    y = atoi(argv[3]);
    printf("%d + %d = %d\n", x, y, add(clnt, x, y));
    printf("%d - %d = %d\n", x, y, sub(clnt, x, y));
    return 0;
}
```

Essas é uma modificação possível para o propósito desse tutorial, mas não é a única maneira de ser feita. Por outro lado, nas modificações no servidor (arquivo **calcula_server.c**) são as seguintes:

```
#include "add.h"

int *
add_100_svc(operandos *argp, struct svc_req *rqstp)
```

```c
{
    static int  result;

    printf("Somando %d e %d \n", argp->x, argp->y);
    result = argp->x + argp->y;

    return &result;
}

int *
sub_100_svc(operandos *argp, struct svc_req *rqstp)
{
    static int  result;

    printf("Somando %d e %d \n", argp->x, argp->y);
    result = argp->x - argp->y;

    return &result;
}
```

Uma vez alterados esses arquivos é preciso compilar todos esses arquivos e gerar os binários que vão rodar no lado cliente e no lado servidor. Em termos de construção esses binários são construídos de acordo com a tabela abaixo:

Nome do Binário	Arquivos que participam da compilação
calcula_client	calcula_client.c, calcula_clnt.c, calcula_xdr.c e calcula.h
calcula_server	calcula_server.c, calcula_svc.c, calcula_xdr.c e calcula.h

A geração desses arquivos binários é auxiliada pelo uso do utilitário make que, por sua vez trabalha em cima do arquivo Makefile.calcula que foi também produzido pelo rpcgen. Portanto, para compilar esse binários dois passos são necessários depois de realizar a alteração dos códigos cliente e servidor:

- Alterar o nome do arquivo Makefile.calcula para Makefile
- Executar o comando make. A partir daqui os códigos binários deverão estar gerados se não houver erro de compilação

Em termos de execução, é importante seguir a ordem de execução abaixo (que vale para qualquer outra execução RPC):

- Colocar o **rpcbind** para executar, caso o mesmo já não esteja (sudo rpcbind)
- Colocar o processo servidor para executar (***#./calcula_server***)
- Colocar o processo cliente para executar passando para ele, dentre outros parâmetros, a identificação do servidor. (***#./calcula_client server N1 N2 <enter>***) onde server é o nome ou endereço IP da máquina onde o servidor está executando; N1 e N2 são os números a serem passados como parâmetro para as funções ADD e SUB.