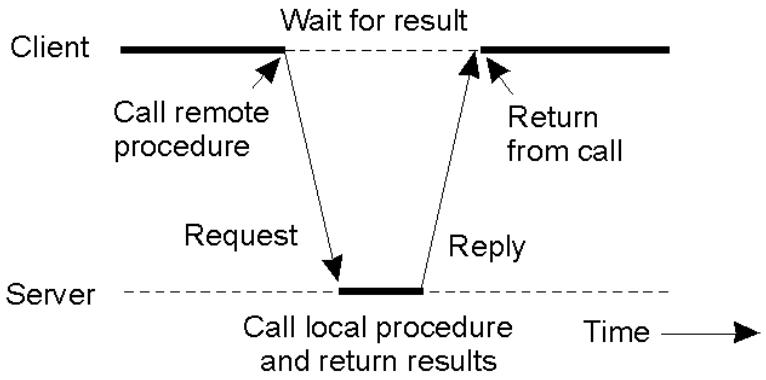
Programação C/S Usando RPC – Chamada de Procedimento Remoto

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:



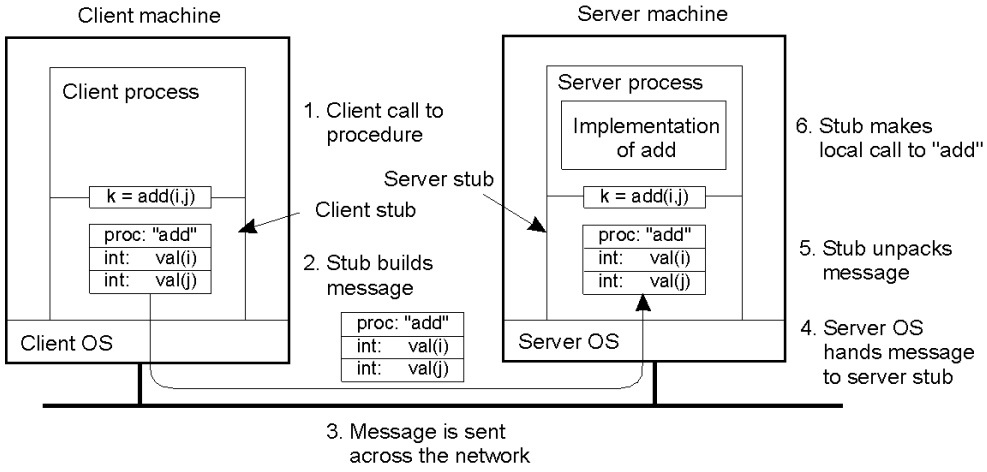
Diante do que foi dito, pode o leito fazer alguns questionamentos tais como:

- 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 (ítem 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

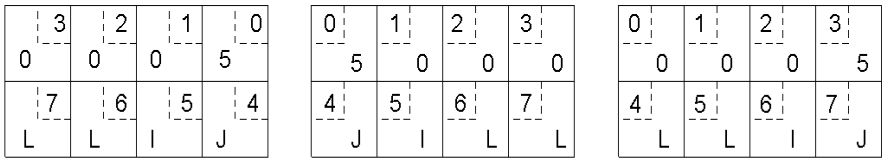
Apesar de todos esses passos e mecanismos de encapsulamento, pricipalmente considerando as funções de rede, a programação RPC é viável pelos seguintes motivos:

- 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 bits	Inteiro binário sinalizado de 32 bits
Unsigned int	32 bits	Inteiro binário não sinalizado de 32 bits
bool	32 bits	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 bits	Inteiro sinalizado de 64 bits
Unsigned hyper	64 bits	Inteiro não sinalizado de 64 bits

[1] Na verdade a programação RPC não consegue ser totalmente transparente para o programador por que nos códigos (principalmente do cliente) é comum perceber referências a funções da biblioteca RPC.

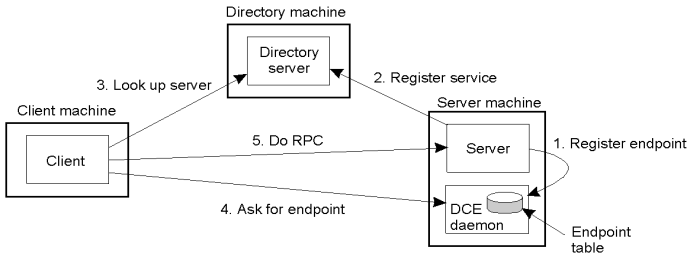
float	32 bits	Numero de ponto flutuante com precisão simples
Doublé	64 bits	Número de ponto flutuante com precisão dupla
Opaque	Arbitrário	Dado não convertido, i.e, dado na representação nativa do remetente
string	Arbitrário	String de caracteres
Fixed array	Arbitrário	Um array de tamanho fixo de qualquer 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 union	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 Constant	Arbitrário	Uma constante simbólica e um valor associado
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	--	Não 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 pode-se concluir que no Linux, para executar qualquer servidor RPC será preciso antes executar o *portmapper*.

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[2].

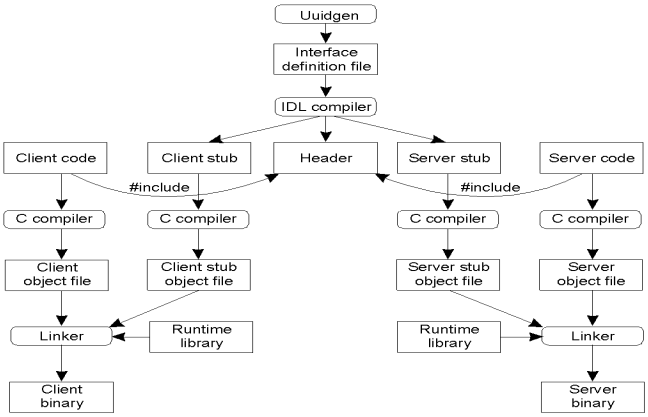
Um servidor RPC provê um grupo de *procedures* ou funções as quais o cliente pode invocar enviando uma requisição para o servidor. O servidor então invoca a *procedure* mencionada pelo cliente, retornando um valor quando necessário. A coleção de *procedures* que um servidor RPC provê é chamada de um 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:

```
# RPC program number data base
# $Revision: 1.5 $        (from 88/08/01 4.0 RPCSRC)
Portmapper              100000          portmap sunrpc
Rstatd                  100001          rstat rup perfmeter
Rusersd                 100002          Rusers
Nfs                     100003          Nfsprog
Ypserv                  100004          Ypprog
Mountd                  100005          mount showmount
Walld                   100008          rwall shutdown
```

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

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.

[2] No caso do Linux, uma vez que servidores RPC podem ser ativados pelo inetd, o portmap deve ser executado antes que o próprio inetd seja ativado.

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 *portmapper* 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 a figura abaixo:

```
1  #include <stdio.h>
2
3  int add (int x, int y) {
4    int result;
5
6    printf("Requisicao de adicao para  %d e %d\n", x, y);
7    result = x + y;
8    return(result);
9  } /* fim funcao add */
10
11 int sub (int x, int y) {
12   int result;
13
14   printf("Requisicao de subtracao para  %d e %d\n", x, y);
15   result = x - y;
16   return(result);
17 } /* fim funcao sub */
18
19 int main( int argc, char *argv[]) {
20   int x,y;
21
22   if (argc!=3) {
23     Fprintf(stderr,"Uso correto: %s num1 num2\n",argv[0]);
24     exit(0); }
25
26   /* Recupera os 2 operandos passados como argumento */
27   x = atoi(argv[1]);   y = atoi(argv[2]);
28   printf("%d + %d = %d\n",x,y, add(x,y));
29   printf("%d - %d = %d\n",x,y, sub(x,y));
30   return(0);
31
32 } /* 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 na Figura abaixo:

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

Nessa figura, 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 de uma 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.h.

calcula_client.c – Nesse arquivo pode-se destacar as seguintes características:

A tabela abaixo contém o código exemplo de calcula_client.c:

```
1  #include "calcula.h"
2
3  Void prog_100(char *host) {
4  CLIENT *clnt;
5  Int  *result_1;
6  Operandos  add_100_arg;
7  Int  *result_2;
8  Operandos  sub_100_arg;
9
10 #ifndef DEBUG
11 clnt = clnt_create (host, PROG, VERSAO, "udp");
12
13 #endif /* DEBUG */
14
15 result_1 = add_100(&add_100_arg, clnt);
16 result_2 = sub_100(&sub_100_arg, clnt);
17
18 #ifndef DEBUG
19 clnt_destroy (clnt);
20 #endif  /* DEBUG */
21 }
22 int main (int argc, char *argv[]) {
23 char *host;
24
25 if (argc < 2) {
26 printf ("usage: %s server_host\n", argv[0]);
27 exit (1); }
28 host = argv[1];
29 prog_100 (host);
30 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:

```
1  #include <memory.h> /* para vso do memset */
2  #include "calcula.h"
3
4  /* Timeout default - pode ser mudado usando clnt_control() */
5  Static struct timeval TIMEOUT = { 25, 0 };
6
7  Int * add_100(operandos *argp, CLIENT *clnt) {
8         static int clnt_res;
9
10 memset((char *)&clnt_res, 0, sizeof(clnt_res));
11 if (clnt_call (clnt, ADD, (xdrproc_t) xdr_operandos, (caddr_t) argp,
12                (xdrproc_t) xdr_int, (caddr_t) &clnt_res,
13                TIMEOUT) != RPC_SUCCESS) {
14                return (NULL); }
15 return (&clnt_res);
16 } /* fim do função add do stub cliente */
17
18 int * sub_100(operandos *argp, CLIENT *clnt) {
19        static int clnt_res;
20
21 memset((char *)&clnt_res, 0, sizeof(clnt_res));
22 if (clnt_call (clnt, SUB, (xdrproc_t) xdr_operandos, (caddr_t) argp,
23        (xdrproc_t) xdr_int, (caddr_t) &clnt_res,
24        TIMEOUT) != RPC_SUCCESS) {
25        return (NULL); }
26 return (&clnt_res);
27 } /* fim da função sub do stub cliente */
```

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.

```
1  #include "calcula.h"
2
3  bool_t xdr_operandos (XDR *xdrs, operandos *objp)
4  {
5          register int32_t *buf;
6
7           if (!xdr_int (xdrs, &objp->x))
8                   return FALSE;
9
10          if (!xdr_int (xdrs, &objp->y))
11                  return FALSE;
12         return TRUE;
13 } /*fim da função xdr_operandos */
```

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.

```
1  #include "calcula.h"
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <rpc/pmap_clnt.h>
5  #include <string.h>
6  #include <memory.h>
7  #include <sys/socket.h>
8  #include <netinet/in.h>
9
10 #ifndef SIG_PF
11 #define SIG_PF void(*)(int)
12 #endif
13
14 static void prog_100(struct svc_req *rqstp, register SVCXPRT *transp)
15 {
16         union {
17                 operandos add_100_arg;
18                 operandos sub_100_arg;
19         } argument;
20         char *result;
21         xdrproc_t _xdr_argument, _xdr_result;
22         char *(*local)(char *, struct svc_req *);
23
24         switch (rqstp->rq_proc) {
25         case NULLPROC:
26                 (void) svc_sendreply (transp, (xdrproc_t) xdr_void, (char *)NULL);
27                 return;
28
29         case ADD:
30                 _xdr_argument = (xdrproc_t) xdr_operandos;
31                 _xdr_result = (xdrproc_t) xdr_int;
32                 local = (char *(*)(char *, struct svc_req *)) add_100_svc;
33                 break;
34
35         case SUB:
36                 _xdr_argument = (xdrproc_t) xdr_operandos;
37                 _xdr_result = (xdrproc_t) xdr_int;
38                 local = (char *(*)(char *, struct svc_req *)) sub_100_svc;
39                 break;
```

```
40
41          default:
42                  svcerr_noproc (transp);
43                  return;
44          }
45          memset ((char *)&argument, 0, sizeof (argument));
46          if (!svc_getargs (transp, _xdr_argument, (caddr_t) &argument)) {
47                  svcerr_decode (transp);
48                  return;
49          }
50          result = (*local)((char *)&argument, rqstp);
51          if (result != NULL && !svc_sendreply(transp, _xdr_result, result)) {
52                  svcerr_systemerr (transp);
53          }
54          if (!svc_freeargs (transp, _xdr_argument, (caddr_t) &argument)) {
55                  fprintf (stderr, "%s", "incapaz de liberar argumentos");
56                  exit (1);
57          }
58          return;
59  } /* fim da função prog_100 */
60
61  int main (int argc, char **argv)
62  {
63          register SVCXPRT *transp;
64
65          pmap_unset (PROG, VERSAO);
66
67          transp = svcudp_create(RPC_ANYSOCK);
68          if (transp == NULL) {
69                  fprintf (stderr, "%s", "não pôde criar o serviço udp.");
70                  exit(1);
71          }
72          if (!svc_register(transp, PROG, VERSAO, prog_100, IPPROTO_UDP)) {
73                  fprintf (stderr, "%s", "incapaz de registrar (PROG, VERSAO, udp).");
74                  exit(1);
75          }
76
77          transp = svctcp_create(RPC_ANYSOCK, 0, 0);
78          if (transp == NULL) {
79                  fprintf (stderr, "%s", "não pôde criar o serviço tcp");
80                  exit(1);
81          }
82          if (!svc_register(transp, PROG, VERSAO, prog_100, IPPROTO_TCP)) {
83                  fprintf (stderr, "%s", "incapaz de registrar (PROG, VERSAO, tcp).");
84                  exit(1);
85          }
86
87          svc_run ();
88          fprintf (stderr, "%s", "svc_run retornou");
89          exit (1);
90          /* NÃO ALCANÇADO */
91  } /* fim do programa principal */
```

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.

```
1  #include "calcula.h"
2
3  int * add_100_svc(operandos *argp, struct svc_req *rqstp)
4  {
5          static int  result;
6           /* INSIRA AQUI O CÓDIGO DESSA FUNÇÃO */
7          return &result;
8  } /* fim da função add remota */
9
10 int * sub_100_svc(operandos *argp, struct svc_req *rqstp)
11 {
12         static int  result;
13          /* INSIRA AQUI O CÓDIGO DESSA FUNÇÃO */
14         return &result;
15 } /* fim da função sub remota */
```

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:

```
1  #include <stdio.h>
2  #include "calcula.h"  /* Criado pelo rpcgen  */
3  int add( CLIENT *clnt, int x, int y) {
4    operandos ops;
5    int *result;
6
7    ops.x = x;     ops.y = y;
8    /* Chama o stub cliente criado pelo rpcgen */
9    result = add_100(&ops,clnt);
10   if (result==NULL) {
11     fprintf(stderr,"Problema na chamada RPC\n");
12     exit(0);  }
13   return(*result);
14 } /* fim função add local */
15
16 int sub( CLIENT *clnt, int x, int y) {
17   operandos ops;
18   int *result;
19   /* Preenche struct operandos p ser enviada ao outro lado */
20   ops.x = x;
21   ops.y = y;
22   /* Chama o stub cliente criado pelo rpcgen */
23   result = sub_100(&ops,clnt);
24   if (result==NULL) {
25     fprintf(stderr,"Problema na chamada RPC\n");
26     exit(0);
27   }
28   return(*result);
29 } /* fim funcao sub local */
30
31 int main( int argc, char *argv[]) {
32   CLIENT *clnt;
33   int x,y;
34
45   if (argc!=4) {
46     fprintf(stderr,"Uso: %s hostname num1 num2\n",argv[0]);
47     exit(0); }
48   clnt = clnt_create(argv[1], PROG, VERSAO, "udp");
39   /* Garantindo a criacao da ligacao com o remoto */
40   if (clnt == (CLIENT *) NULL) {
41     clnt_pcreateerror(argv[1]);
42     exit(1); }
43   /* Recupera os 2 operandos passados como argumento */
44   x = atoi(argv[2]);  y = atoi(argv[3]);
45   printf("%d + %d = %d\n",x,y, add(clnt,x,y));
46   printf("%d - %d = %d\n",x,y, sub(clnt,x,y));
47   return(0);
48 } /* fim main */
```

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:

```
1  int * add_100_svc(operandos *argp, struct svc_req *rqstp) {
2          Static int  result;
3
4          Printf("Requisicao de adicao para  %d e %d\n", argp->x, argp->y);
5          Result = argp->x + argp->y;
6          Return (&result);
7  } /* fim funcao add remota */
8
9
10 int * sub_100_svc(operandos *argp, struct svc_req *rqstp) {
11         static int  result;
12
13         printf("Requisicao de subtracao para  %d e %d\n\n", argp->x, argp->y);
14         result = argp->x - argp->y;
15         return (&result);
16 } /* fim funcao sub remota */
```

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 portmapper para executar, caso o mesmo já não esteja (/usr/sbin/portmap)
- 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.

Exercícios:

1) Proceda as seguintes alterações no programa acima:

a. Altere o número e versão do programa e os números das funções no arquivo calcula.x, execute o rpcgen para que novos arquivos sejam gerados e veja que modificações ocorreram nos nomes das funções em cada um dos arquivos. Gere novos binários e coloque o módulo servidor para executar. Verifique com o comando rpcinfo –p se novo número de program e versão estão registrados no rpcgen

b. Altere o programa acima inserindo as funções MULT e DIV, responsáveis por retornar, respectivamente o produto e a divisão entre os dois números informados pelo usuário. Obs.: Considerar N1 e N2 > 0.

2) Verifique o procedimento ulookup.c e altere-o para que funcione de acordo com a arquitetura RPC Cliente/Servidor. Verifique a localização desse arquivo com o professor. Monte um relatório contendo o código do cliente e do servidor e as modificações feitas, incluindo o arquivo IDF usado para o rpcgen (.x)

3) Faça um procedimento RPC Cliente/Servidor no qual o cliente passe para o servidor uma frase a ser apresentada na tela pelo servidor. Verifique o arquivo hello.x para referência. Monte um relatório contendo o código original, a interface IDF justificando o tipo de dados, e os códigos do cliente e servidor alterado.

6. Alguns detalhes sobre a Interface de Definição de tipos XDR

As declarações XDR são muito similares ao conjunto de declarações C, e por esse motivo muitos se confundem com a definição de tipos de dados em XDR. XDR e C são linguagens diferentes e existem vários tipos de dados, ordenação de bytes e alinhamentos que um arquivo de cabeçalho em C não necessita se preocupar.

Algumas características da linguagem XDR:

Existem vários tipos básicos de inteiros (char, short, int, long) com versões com sinal (signed) e sem sinal (unsigned).

Existem várias definições de números reais de precisão simples (float) ou dupla (double)

Existe o tipo void para especificar vazio

Existem apontadores para construir estruturas dinâmicas.

Os tipos enumerados do XDR são tipos verdadeiros (diferente do C no qual tipos enumerados são mera enumeração de constantes). No caso de XDR os tipos enumerados são arranjos discretos de variáveis (mesmo que elas não sejam checadas...)

XDR define alguns tipos extras tais como booleanos (bool), strings (string) e tipos opacos (opaque)

Provê alguma extensão para as construções do padrão C tais como unions discriminadas.

Alguns exemplos XDR:

a)

```
enum prioridade {                    struct alarme {
        baixa   = 1,                         prioridade      prio;
        media = 2,                           string          texto<>;
        alta    = 3                  };
};
```

O segmento acima define um tipo composto de nome alarme com dois campos: (i) prioridade, o qual é uma enumeração com três valores possíveis, e (ii) uma variável texto do tipo string que, na verdade em C seria um apontador para um tipo char.

b)

```
const TAMMAX = 256;
typedef string nomearq<TAMMAX>;
sctruct arquivo {
    nomearq   nome;
    opaque    dado<>;
};
```

É possível que os conteúdos do arquivo não estejam claramente estruturados ou então talvez seja necessário especificar a estrutura exata contida nele. Neste caso o tipo opaco com um tamanho variável (tamanho variável é indicado pelos sinais de maior e menor juntos (<>)) e sem tamanho limite máximo. É possível especificar um tamanho fixo entre os sinais de maior e menor (<>) mas, obviamente isso não faz sentido nesse exemplo. O XDR irá preservar a ordem dos bytes do arquivo em ambas as máquinas de comunicação, tomando a responsabilidade da aplicação ler os dados corretamente para cada arquitetura. No que constantes (consts) em XDR (diferente do que é feito em C) podem ser usadas para especificar dimensões de arrays (eles são traduzidos dentro das constantes do pré-processador).

c) Uniões Discriminadas

XDR provê uma extensão para o tipo union da linguagem C e nesse caso, essa extensão é chamada de união discriminada. O exemplo abaixo mostra como funcionam esses tipos:

```
enum TipoRetorno {
        NOME_RES   = 1,
        NOME_COD  = 2
};
union Result switch (TipoRetorno tipo) {
   case  NOME_RES:
        string   nome<>;
```

```
    case  NOME_COD:
            int        code;
    default:
            void;
};
```

Essa descrição diz que Result tem um campo chamado tipo e, dependendo do seu valor, ele também terá um campo nome ou um campo code ou nenhum campo a mais. Ao executar o rpcgen sobre este exemplo ficará claro que a struct nomeada Result com uma union dentro e será chamada de Result_u. No entanto é preciso lembrar a ordem de declaração dos campos para acessar cada tipo de estrutura da linguagem C.

d) Estruturas alocadas dinamicamente

Veja o exemplo a seguir:

```
struct lista {
        lista     * prox;
        string  dado<>;
};
typedef lista * ap_lista;
```

```
program RPCDEMO {
        version RPCDEMO_VER {
            void RPCDEMO_PRINT (ap_lista) = 1;
        } = 1;
} = 60000;
```

O exemplo é fácil de entender mas existem dois detalhes importantes a serem destacados:

- a criação de um tipo de dados typedef para referenciar a estrutura usada na lista encadeada
- Sempre que se usa estruturas dinâmicas existe uma questão a respeito de alocação e liberação de memória. É importante lembrar que nesse caso, o mesmo módulo que alocou memória é também responsável por libera-la.

Em um ambiente distribuído isso é mais difícil por que, se o cliente alocar uma estrutura de dados dinâmica e passa-la para o servidor em uma chamada RPC, então o servidor automaticamente aloca memória para ele e libera a memória após chamar sua implementação (não é possível assumir que argumentos dinâmicos persistirão no código). No entanto, se o servidor retornar uma estrutura dinâmica, então o cliente aloca memória mas não pode liberar nada por que ele não sabe quando essa memória deixará de ser útil. Por esse motivo, é responsabilidade do programador liberar memória que tenha sido usada, através da função xdr_free (xdr_ListPtr, list_ptr). O primeiro parâmetro é a função de conversão que é gerada automaticamente com rpcgen (tente rpcgen –c –N text.x > text_xdr.c) e o segundo argumento é o ponteiro retornado pela chamada de procedimento remoto.

É importante lembrar que isso se aplica a tipos construídos (tais como o tipo string). Mas, neste caso, nenhuma rotina de conversão é gerada, e está na biblioteca do sistema. Como observação, procure não usar xdr_string, mas sim xdr_wrapstring. A razão para essa escolha fica como um exercício.

Não é possível usar o opaque como um tipo para qualquer argumento em uma RPC. Veja um exemplo de declaração INCORRETA:

```
typedef opaque op<>;
program DEMO {
      version  DEMO_1 {
          void DEMO_PROC_1(op o) = 1;
     } = 1;
} = 60000;
```

```
/* definição incorreta no XDR: */
program DEMO2 {
      version DEMO2_1 {
           void  DEMO2_1_PROC_1 (opaque o<>) = 1;
      } = 1;
} = 60001;
```

Verifique o código gerado para a versão correta e então examine a declaração xdr_string e xdr_wrapstring em /usr/include/rpc/xdr.h para perceber as inconsistências.

Especificação do Protocolo

Conforme já foi demonstrado anteriormente, o protocolo entre cliente e servidor é especificado como um conjunto de procedimentos com um valor de retorno opcional e qualquer número de argumentos. Cada conjunto de argumentos é agrupado junto em uma versão de protocolo, com um número consistente e todas as versões são agrupadas juntas em um programa RPC. A figura abaixo mostra um exemplo:

```
program PROG {
    version PROG1 {
            void   PROG_PROC1 (int a) = 1;
            void   PROG_PROC2 (string str<>) = 2;
    } = 1;
    version PROG2 {
             int   PROG_PROC1 (int a, int b)      = 1;
             int   PROG_PROC2  (string  str<>)  = 2;
    } = 2;
} = 60000;
```

Números de programa e nomes devem ser únicos em uma versão, números de versão e identificadores devem ser únicos em cada programa, e cada programa e identificador deve ser único no sistema como um todo.

Todo o dado necessário para executar uma chamada em um programa cliente pode ser transmitida usando um datagrama UDP. Como esse protocolo é não confiável, existem riscos de duplicadas chegarem do outro lado. Existem outras situações que causam réplicas de datagramas do outro lado. A implementação de RPC da SUN não cuida de duplicatas, então o programador deve finalizar com múltiplas chamadas para uma dada função ou procedure quando ele apenas gostaria de fazer uma única chamada. Isto é chamada de semântica “least once” (pelo menos uma vez). Quer dizer que não há garantias que a procedure remota chamada será chamada apenas uma vez.

Por essa razão todos os protocolos devem ser especificados de maneira que múltiplas chamadas para uma procedure com os mesmos argumentos retornem exatamente os mesmos resultados. Esses são chamados protocolos idempotentes. Por exemplo, no caso da leitura de um arquivo, a procedure chamada deve retornar ou o arquivo inteiro ou especificar o offset a partir do qual os dados devem ser lidos.

Ao usar TCP existe garantia de liberação e integridade de seqüências de dados e não há necessidade de preocupação com os problemas que ocorrem quando se usa UDP.