Universidade Federal
do Espírito Santo

CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA

Processamento Paralelo e Distribuído – Turmas 01 e 02 (EARTE) – 2022/1
Prof. Rodolfo da Silva Villaça – rodolfo.villaca@ufes.br
Laboratório II – Chamada de Procedimento Remoto (RPC)

1. Objetivos

- Experimentar a implementação de sistemas distribuídos baseados na arquitetura Cliente/Servidor usando o conceito de Chamada de Procedimento Remoto (RPC) nas linguagens C e Python;
- Alternativamente, explorar a implementação do mesmo sistema usando Java e Invocação de Método Remoto (RMI).

2. Conceitos Básicos

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) são consideradas um pilar básico para a implementação de boa parte dos Sistemas Distribuídos. 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.

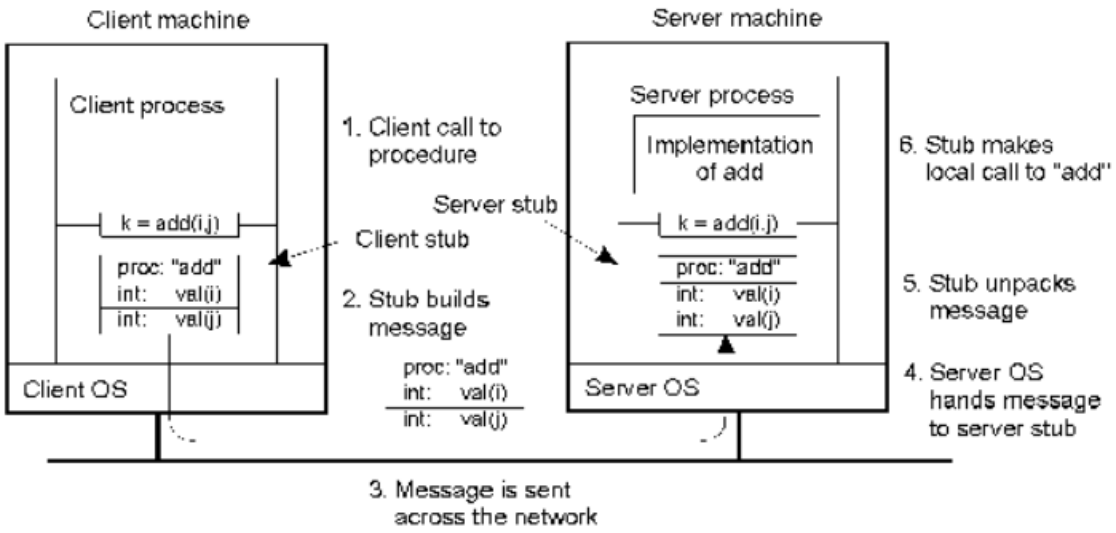
Diante do que foi dito, surgem os seguintes questionamentos:

- 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 roteiro apresenta tópicos que incluem uma parte teórica e um exemplo de conversão de um programa convencional em um programa RPC.

Princípios da comunicação RPC entre Cliente e 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 do stub, 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.

2.1. Linguagem C

Apesar de todos esses passos e mecanismos de encapsulamento, principalmente considerando as funções de rede, a programação RPC é complexa pela necessidade encapsulamento e geração de stubs, arquivos de include e todas as chamadas de rede. Com a biblioteca de RPC, essas tarefas deixam de ser 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.

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 resolver esse problema 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
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_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)

Como o Cliente Localiza o Servidor RPC?

Toda aplicação RPC cliente/servidor 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.

O serviço de diretório 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. 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 contatar 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. Esse arquivo só deve ser alterado se um novo serviço RPC for introduzido no servidor.

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

Junto com a biblioteca RPC existe uma ferramenta denominada rpcgen, cujo objetivo é gerar, a partir de um arquivo de definição de interface (IDF, ou Interface Definition File), todos os módulos de software que devem estar nos lados cliente e servidor, incluindo os stubs. De um modo geral 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 em um arquivo de definição de interface (Interface Definition File). 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. Essas informações devem ser incluídas no arquivo de interface IDF e, a partir dele, gera-se todos os códigos necessários.
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, considerado um padrão de fato no mercado. Esse utilitário pressupõe que o arquivo IDF tem um nome com sufixo “.x”, e com base nele, os códigos são gerados em 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 nesses 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 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 noservidor, 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. 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 endereço IP de localização do servidor.

Exemplo de construção de um programa 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. No caso de uma aplicação convencional, esse código poderia ser dividido em programa principal e duas funções: add e sub, conforme abaixo:

simpleCalc.c

```
#include <stdio.h>
#include <stdlib.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 */
```

Gerar o arquivo de definição de Interface (IDF) a partir do código apresentado

A ideia 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. 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, rpcCalc.x e terá o conteúdo especificado a seguir:

rpcCalc.x

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

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

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 traz 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 rpcCalc.x

Os arquivos gerados serão os seguintes:

Arquivo	Significado
rpcCalc.h	Arquivo com as definições que deverão estar inclusas nos códigos cliente e servidor
rpcCalc_client.c	Arquivo contendo o esqueleto do programa principal do lado cliente
rpcCalc_cInt.c	Arquivo contendo o stub do cliente
rpcCalc_xdr.c	Contém as funções xdr necessárias para a conversão dos parâmetros a serem passados entre hosts
rpcCalc_svc.c	Contém o programa principal do lado servidor.
rpcCalc_server.c	Contém o esqueleto das rotinas a serem chamadas no lado servidor
Makefile.rpcCalc	Deve ser renomeado para Makefile. Contém as diretivas de compilação para a ferramenta make.

- **rpcCalc.h:** 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 “.x”.

- **rpcCalc_client.c:** 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:
  a) Chama a função clnt_create cujo objetivo é contatar e gerar uma conexão com o portmapper instalado no host remoto e questionando a ele qual é o número da porta do servidor RPC;
  b) 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.

- **rpcCalc_clnt.c:** Contém as funções referenciadas no código rpcCalc_client.c, ou seja, as funções add_100 e sub_100. Essas funções realizam os seguintes passos:
  a) Encapsulamento das chamadas do xdr através da chamada à função xdr_operandos, que por sua vez, está presente no arquivo rpcCalc_xdr.c. Essa chamada é feita para que haja o encapsulamento das variáveis embutidas na estrutura operandos no formato do xdr e, como pode ser visto, está embutida na chamada à system call clnt_call, descrita a seguir;
  b) 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: i) a identificação do servidor remoto (variável clnt); ii) identificação da função remota a ser chamada nesse servidor (variável ADD ou SUB); iii) parâmetros convertidos no formato XDR e, iv) uma variável para conter o resultado ofertado pela função remota chamada.

- **rpcCalc_xdr.c:** Contém as funções xdr a serem usadas pelo código, conforme apresentado anteriormente nas tabelas de tipos e funções nesse documento. 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;

- **rpcCalc_svc.c:** 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 essa identificação. Dentre outras funcionalidades, o programa principal faz o seguinte:

a) 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;
b) Cria sockets UDP e TCP associando-os às portas livres no servidor (svcudp_create e svctcp_create);
c) Registra essas portas no portmapper (funçãosvc_register) vinculadas ao númerodo 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;
d) 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>, por exemplo.

A função prog_100, por sua vez realiza as seguintes funções:
a) Recebe os parâmetros de entrada, dentre eles a identificação da função (ADD ou SUB) chamada;
b) Chama a função xdr_operandos para realizar o processo inverso de conversão, ou seja, do formato XDR para o formato local desse servidor;
c) Chama uma das funções (add_100_svc ou sub_100_svc) presentes no arquivo rpcCalc_server.c;
d) Recebe o retorno da função chamada e converte esses valores novamente no formato xdr, através da chamada de funções dessa biblioteca;
e) Chama a funçãos endreply que retorna para o stub do cliente os resultados calculados pelas funções add_100_svc ou sub_100_svc.

- **rpcCalc_server.c:** Contém os códigos das funções que devem ser alterados para realizar o que se deseja. 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 rpcCalc_client.c.

Um exemplo de modificação possível no arquivo rpcCalc_client.c está a seguir:

rpcCalc_client.c

```
#include <stdio.h>
#include <stdlib.h>
#include "rpcCalc.h" /* Criado pelo rpcgen */

int add(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 = 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;

   if (argc!=4) {
      fprintf(stderr,"Uso: %s hostname num1 num2\n",argv[0]);
      exit(0);
   }
   clnt = clnt_create(argv[1], PROG, VERSAO, "udp");

   /* 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);
} /* fim main */
```

Essa é uma possível modificação, mas não é a única maneira de ser feita. Por outro lado, uma modificação possível no lado servidor (arquivo rpcCalc_server.c) é a seguinte:

rpcCalc_server.c

```
#include "rpcCalc.h"

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

      printf("Requisicao de adicao para %d e %d\n", argp->x, argp->y);
      result = argp->x + argp→y;
      return &result;
} /* fim funcao add remota */

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

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

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

Programa	Arquivos Fonte
rpcCalc_client	rpcCalc_client.c, rpcCalc_clnt.c, rpcCalc_xdr.c e rpcCalc.h
rpcCalc_server	rpcCalc_server.c, rpcCalc_svc.c, rpcCalc_xdr.c e rpcCalc.h

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

a) Alterar o nome do arquivo Makefile.rpcCalc para Makefile;

b) 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):

a) Colocar o portmapper para executar, caso o mesmo já não esteja;
b) Colocar o processo servidor para executar;
c) Colocar o processo cliente para executar passando para ele, dentre outros parâmetros, o endereço do servidor.

2.2. Python

XML-RPC é um método de chamada de procedimento remoto que usa XML passado via HTTP como um transporte. Com ele, um cliente pode chamar métodos com parâmetros em um servidor remoto (o servidor é nomeado por um URI) e receber dados estruturados. xmlrpc é um pacote que coleta módulos de servidor e de cliente que implementam o XML-RPC.

Usaremos a biblioteca xmlrpc[1] e a seguir seguem alguns destaques importantes da biblioteca:

- **xmlrpc.server.SimpleXMLRPCServer:** Cria uma nova instância do servidor. Esta classe fornece métodos para registro de funções que podem ser chamadas pelo protocolo XML-RPC.
- **xmlrpc.server.SimpleXMLRPCServer.register_function:** Registra uma função que possa responder às solicitações XML-RPC. Se name for fornecido, será o nome do método associado a function.
- **xmlrpc.client.ServerProxy:** Uma instância ServerProxy gerencia a comunicação com um servidor XML-RPC remoto.

3. Atividade

Arquivos fornecidos com o Laboratório II:

Arquivo	Significado
SimpleCalc.c	Código-fonte em C da calculadora com execução local, sem RPC

1https://docs.python.org/pt-br/3/library/xmlrpc.html

rpcCalc.x	Arquivo de Definição de Interface da Calculadora RCP (rpcCalc)
rpcCalc_client.py	Cliente rpcCalc em Python usando XML-RPC
rpcCalc_server.py	Servidor rpcCalc em Python usando XML-RPC

Com base nestes exemplos, você precisará construir um protótipo similar a um minerador de criptomoedas do tipo bitcoin, porém no modelo cliente/servidor baseado em RPC, com implementação em Python e C. Cada implementação deverá ter seu próprio cliente e servidor, não há exigência (por enquanto) de interoperabilidade.

O servidor deverá ter o seguinte funcionamento:

a) Manter, enquanto estiver em execução, uma tabela com os seguintes registros:

TransactionID	Challenge	Seed	Winner
int	int	str	int

- TransactionID: Identificador da transação, representada por um valor inteiro (32 bits). Use valores incrementais começando em 0 (zero);
- Challenge: Valor do desafio criptográfico associado à transação, representado por um número [1..128], onde 1 é o desafio mais fácil. Gere desafios aleatórios, ou sequenciais, experimente as diferentes abordagens;
- Seed: String que, se aplicada a função de hashing SHA-1, solucionará o desafio criptográfico proposto;
- Winner: ClientID do usuário que solucionou o desafio criptográfico para a referida TransactionID (mesma linha da tabela). Enquanto o desafio não foi solucionado, considere que o ClientID = -1;

b) Ao carregar, o servidor deverá gerar um novo desafio com transactionID=0;
c) Disponibilizar as seguintes chamadas de procedimento remoto para os clientes RPC:

Nome	Parâmetros	Significado
getTransactionID()	None	Retorna o valor atual <int> da transação com desafio ainda pendente de solução.
getChallenge()	int transactionID	Se transactionID for válido, retorne o valor

		do desafio associado a ele. Retorne -1 se o transactionID for inválido.
getTransactionStatus()	int transactionID	Se transactionID for válido, retorne 0 se o desafio associado a essa transação já foi resolvido, retorne 1 caso a transação ainda possua desafio pendente. Retorne -1 se a transactionID for inválida.
submitChallenge()	int transactionID, int ClientID, int seed	Submete uma semente (seed) para o hashing SHA-1 que resolve o desafio proposto para a referida transactionID. Retorne 1 se a seed para o desafio for válido, 0 se for inválido, 2 se o desafio já foi solucionado, e -1 se a transactionID for inválida.
getWinner()	int transactionID	Retorna o clientID do vencedor da transação transactionID. Retorne 0 se transactionID ainda não tem vencedor e -1 se transactionID for inválida.
getSeed()	int transactionID	Retorna uma estrutura de dados (ou uma tupla) com o status, a seed e o desafio associado à transactionID.

d) O servidor deve ficar em loop atendendo diversas conexões, só podendo ser interrompido com um <ctrl+c> ou similar.

O cliente, por sua vez, deve receber como parâmetro o endereço do server e um menu com as seguintes opções:

Item do Menu	Ação
getTransactionID	Solicita ao servidor a transação corrente (atual). Imprime na tela.
getChallenge	Lê um valor de transactionID no teclado e solicita o valor do desafio associado a essa transação ao servidor. Imprime na tela.
getTransactionStatus	Lê um valor de transactionID no teclado e solicita o estado. Imprime na tela. da referida transação ao servidor. Imprime na tela.
getWinner	Lê um valor de transactionID no teclado e solicita o clientID do vencedor. Imprime na tela.
GetSeed	Lê um valor de transactionID no teclado e solicita a seed associada a essa transactionID, se o desafio já foi resolvido. Imprime na tela.
Minerar	1. Buscar transactionID atual; 2. Buscar challenge (desafio) associada ao transactionID atual;

	3. Buscar, localmente, uma seed (semente) que solucione o desafio proposto – sugere-se usar múltiplas threads para isso!!!!!! 4. Imprimir localmente a seed encontrada; 5. Submeter resposta ao servidor e aguardar resultado; 6. Imprimir/Decodificar resposta do servidor.

4. Instruções Gerais

1. O trabalho pode ser feito em grupos de 2 ou 3 alunos: não serão aceitos trabalhos individuais ou em grupos de mais de 3 alunos;
2. Os grupos poderão implementar os trabalhos usando qualquer uma dentre as três linguagens de programação: C, Java ou Python;
3. Data de Entrega: 31/05/2022;
4. A submissão deverá ser feita por meio de um *link* com a disponibilização dos códigos no Github, em modo de acesso público ou, minimamente, que meu e-mail rodolfo.villaca@ufes.br tenha direito de acesso ao código;
5. A documentação deverá ser feita na própria página do Github através do arquivo README[2];
6. O grupo deverá gravar um vídeo de no máximo 5 min apresentando o funcionamento/teste do trabalho.

Bom trabalho!

2 https://guides.github.com/features/wikis/