

CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA

Sistemas Distribuídos – Turmas 01 e 02 – 2023/1

Prof. Rodolfo da Silva Villaça – rodolfo.villaca@ufes.br

Monitor: Eduardo M. Moraes Sarmento – eduardo.sarmiento@ufes.br

Laboratório III – 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);
- Praticar a implementação de uma aplicação de calculadora, usando o conceito de RPC nas linguagens C e Python;

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 existentes atualmente.

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 chamadas ficam residentes em *hosts* distintos. Nesse contexto, um *host* (cliente) executando o programa principal aciona uma chamada de procedimento remoto e ficaria aguardando o resultado da chamada, que será executada no *host* remoto (servidor). O *host* remoto (*servidor*) teria a implementação das referidas funções no seu espaço de memória local e fica aguardando requisições remotas provenientes do cliente. Ao chegar uma requisição, o servidor executa a função identificada e retorna os resultados para o cliente por meio de trocas de mensagens em uma infraestrutura de rede. Diante disto, surgem os seguintes questionamentos:

- Como o cliente consegue passar para o servidor os parâmetros de chamada de procedimento remoto?
- Diante de múltiplas requisições de diferentes clientes, como o servidor consegue individualizar cada chamada?
- Que mecanismos o cliente e o servidor devem possuir para viabilizar chamadas remotas?

CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA

Princípios da comunicação RPC entre Cliente e Servidor

O objetivo de uma 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* (esqueleto). Dessa maneira, o código de uma aplicação RPC terá um comportamento similar ao exemplo que está apresentado na Figura 1.

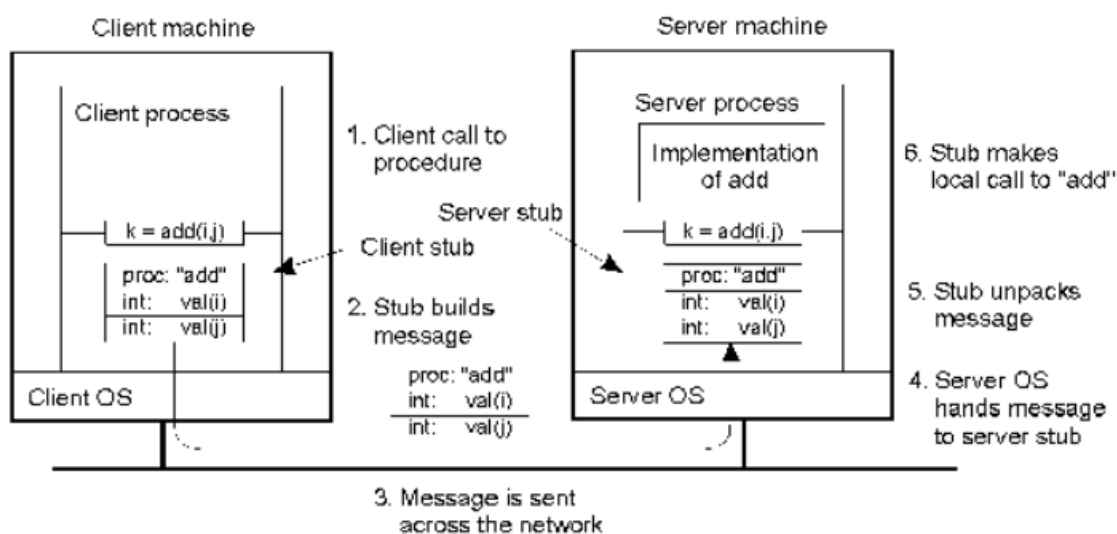


Figura 1 – Funcionamento de uma aplicação RPC.

Perceba que, da forma como está apresentado, existirá um esqueleto (*stub*) associado ao código do cliente (*Client stub*) e outro associado ao código do servidor (*Server stub*). Entenda o *stub* de um procedimento RPC como sendo o equivalente ao *header* de um procedimento local. Dessa forma, o diálogo dos módulos cliente e servidor acontecerá de acordo com a seguinte sequência:

- a) O processo no *host* cliente (*Client machine*) chama um procedimento em um processo remoto (*Client process*) que está implementado no *stub* cliente (*Client stub*);
- b) O *stub* cliente constrói uma mensagem endereçada ao *host* servidor (*Server machine*) usando uma chamada para o seu próprio sistema operacional (*Client OS*);
- c) O sistema operacional do cliente envia a mensagem pela rede o *host* remoto (*Server machine*);

CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA

- d) O sistema operacional do servidor (*Server OS*) recebe a mensagem enviada pela rede e a entrega para o procedimento remoto (*Server stub*) implementado no servidor, mais especificamente no processo servidor (*Server process*);
- e) O *stub* servidor desempacota a mensagem, decodifica os parâmetros da chamada de procedimento e executa o processo no servidor (*Server machine*), mais especificamente, executa o procedimento associado à mensagem recebida do *host* cliente;
- f) Após a execução do procedimento remoto no servidor (*Server process*) o resultado é retornado para o *stub* servidor;
- g) O *stub* servidor empacota os dados retornados em uma mensagem de rede destinada ao *host* cliente e chama o sistema operacional local (*Server OS*);
- h) O sistema operacional do servidor envia a mensagem para o sistema operacional do cliente por meio de uma rede de comunicação de dados;
- i) O sistema operacional do cliente entrega a mensagem recebida para o *stub* cliente;
- j) O *stub* cliente desempacota a mensagem, decodifica os dados e retorna-os para a aplicação cliente que havia iniciado a chamada RPC no item a).

Devido a todos esses passos e mecanismos de encapsulamento, principalmente considerando as funções de rede, a programação RPC é complexa. Algumas linguagens de programação possuem bibliotecas para facilitar a geração de chamadas RPC. Essas bibliotecas contêm ferramentas que auxiliam a geração dos *stubs*, cabendo ao programador apenas alterar os arquivos relacionados à lógica de execução das aplicações cliente e servidor, tornando a programação RPC mais próxima da programação convencional.

O código-fonte dos exemplos usados neste laboratório estão disponíveis no repositório¹ da disciplina, especificamente na pasta *lab3*.

Atividade 1: Construção de um programa RPC em C.

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.

¹ <https://github.com/nerds-ufes/sistdist.git>

CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA

No caso de uma aplicação convencional, *simpleCalc.c*, esse código poderia ser dividido em programa principal e duas funções: *add* e *sub*. A transformação deste programa em uma aplicação RPC tem como passo inicial a geração do arquivo de definição de interface (IDF) a partir do código apresentado. Este arquivo tem como objetivo identificar as funcionalidades que devem estar no programa principal e as sub-rotinas que serão acionadas no servidor para execução remota dessas chamadas.

Retomando o objetivo deste laboratório, a ideia é fazer com que o programa principal seja executado em um processo (cliente) e as funções *add* e *sub* sejam executadas em um outro processo (servidor), e ambos se comuniquem por meio de mensagens de rede em *hosts* distintos. Para isso, a maneira mais eficiente é lançar mão da ferramenta *rpcgen*. Essa ferramenta consegue gerar todos os códigos necessários à implementação dos *stubs* a partir de um arquivo IDF que contém basicamente:

- Definição dos parâmetros que vão ser passados para a(s) chamadas(s) RPC;
- Definição de quais são as funções remotas disponíveis no servidor.

Neste exemplo, a partir do programa *simpleCalc.c*, pode-se gerar um arquivo IDF cujo nome será, nesse exemplo, *rpcCalc.x* e terá o conteúdo especificado a seguir:

```
struct operandos {  
    int x;  
    int y;  
};  
  
program PROG {  
    version VERSAO {  
        int ADD(operandos) = 1;  
        int SUB(operandos) = 2;  
    } = 100;  
} = 5555;
```

No arquivo *rpcCalc.x* 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 chamadas de procedimento remoto. Essa definição segue a recomendação de que a passagem de uma única variável (mesmo que composta) é melhor do que passar muitas variáveis individuais;

CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA

- A definição *program PROG* define o identificador do programa como 5555. Ou seja, os processos cliente e servidor terão um número de identificação RPC igual a 5555 que deverá ser reconhecido tanto no *host* cliente quanto no servidor;
- Além disso, é importante perceber que essa definição traz um número de versão (no caso 100) para esse programa;
- Finalmente, a definição da IDF traz, também, um código de identificação para cada uma das duas rotinas declaradas. Dessa forma, a função *add* terá código 1 e a função *sub* terá o código 2;
- Assim, na mensagem RPC enviada por um cliente para um servidor RPC a tupla (5555, 100, 2) identifica uma chamada à função *sub* (2) na versão 100 do processo servidor *Server stub* (5555).

Vale observar que a notação IDF não é feita em linguagem C, apesar da similaridade. Uma vez gerado esse arquivo, com extensão *.x*, 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. |

CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA

- **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;
- **rpcCalc_client.c:** Este programa contém duas funções: i) o programa principal (main), que recebe o endereço do *host* remoto (servidor) via parâmetro de entrada e, ii) a função *prog_100* que realiza o seguinte:
 - a) Chama a função *clnt_create* para contactar o *host* remoto (servidor) questionando a porta do serviço RPC associada a essa função;
 - b) Chama as funções *add* ou *sub*, presentes no *stub* local, passando como operandos as variáveis *add_100_arg* ou *sub_100_arg*. É importante perceber que após o *rpcgen* essas variáveis não estão inicializadas, e uma das alterações requeridas pelo programador é preencher, aqui, 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 o encapsulamento e envio dos argumentos, no formato XDR, por meio de mensagens para o servidor RPC remoto.
- **rpcCalc_xdr.c:** Contém as funções de conversão para o formato XDR a serem usados no código. Nesse caso a função chamada é a *xdr_int* que converte os inteiros *x* e *y* para o formato *xdr_int*;
- **rpcCalc_svc.c:** Representa o *stub* servidor. Este arquivo possui duas rotinas principais: i) o programa principal, responsável pelos controles iniciais de registro do servidor RPC 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 está instalado no servidor RPC *rpcbind* (função *pmap_unset*);
- b) Cria *sockets* associando-os às portas livres no servidor (*svcudp_create* e *svctcp_create*);
- c) Registra essas portas no *rpcbind* (função *svc_register*) vinculadas ao número do programa, versão e identificador da função que deve ser chamada;

CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA

d) Chama a função *svc_run* para habilitar a escuta permanente da porta que foi associada a esse servidor;

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*);
- b) Chama a função *xdr_operandos* para realizar o processo inverso de conversão, ou seja, do formato XDR para o formato local;
- 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 função *sendreply* que retorna para o *stub* cliente o resultado das funções *add_100_svc* ou *sub_100_svc*.
- **rpcCalc_server.c:** Contém a implementação dos códigos das funções remotas, que devem ser alteradas 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 deve ser feito no arquivo *rpcCalc_client.c*.

Exemplos de modificações possíveis nos arquivos *rpcCalc_client.c* e *rpcCalc_server.c* estão disponíveis no repositório da disciplina.

Uma vez alterado é preciso compilar todos esses arquivos e gerar os binários que vão rodar no lado cliente e no lado servidor. 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, após geração do código-fonte por meio da ferramenta *rpcgen*, e após realizar a alteração dos códigos cliente e servidor, para compilar os programas cliente e servidor, dois passos são necessários:

- a) Alterar o nome do arquivo *Makefile.rpcCalc* para *Makefile*;
- b) executar o comando *make*.

CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA

A partir daqui os códigos binários deverão estar prontos. Em termos de execução, é importante seguir a ordem de execução a seguir:

- a) Colocar o *rpcbind* para executar, caso o mesmo já não esteja em execução;
\$service rpcbind status
- b) Colocar o processo servidor para executar;
- c) Colocar o processo cliente para executar passando para ele, dentre outros parâmetros, o endereço do servidor.

Atividade 2: Construção de um programa RPC em Python usando XML-RPC.

XML-RPC é uma implementação RPC em Python que usa codificação XML para a mensagem e HTTP como protocolo de transporte. Com ele, um cliente pode chamar métodos com parâmetros em um servidor remoto, nomeado por uma URI, e receber dados estruturados. O *xmlrpc* é o pacote Python que implementa os módulos servidor e de cliente RPC usando XML-RPC.

Neste laboratório usaremos a biblioteca *xmlrpc*² e, a seguir, seguem alguns destaques importantes desta biblioteca:

- ***xmlrpc.server.SimpleXMLRPCServer*:** Cria uma nova instância do servidor RPC. Esta classe fornece métodos para registro de funções que podem ser chamadas pelo protocolo XML-RPC.
- ***xmlrpc.server.SimpleXMLRPCServer.register_function*:** Registra uma função para responder às solicitações XML-RPC. Recebe um nome e uma função como parâmetros, e este nome será associado à função no procedimento remoto. Funcionalidade similar ao *rpcbind* em C;
- ***xmlrpc.client.ServerProxy*:** Uma instância da classe *ServerProxy* que gerencia a comunicação com um servidor XML-RPC remoto.

Os seguintes arquivos Python estão fornecidos neste laboratório:

| Arquivo | Significado |
|--------------------------|--|
| <i>rpcCalc_client.py</i> | Cliente <i>rpcCalc</i> em Python usando XML-RPC |
| <i>rpcCalc_server.py</i> | Servidor <i>rpcCalc</i> em Python usando XML-RPC |

²<https://docs.python.org/pt-br/3/library/xmlrpc.html>

CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA

Execute inicialmente a aplicação servidor *rpcCalc_server.py* em um terminal e, posteriormente, execute a aplicação cliente *rpcCalc_client.py* e teste a calculadora na versão Python.

Atividade 3: Construção de um programa RPC em Python usando gRPC.

O gRPC³ traz uma implementação de RPC criada pelo Google, de código aberto e alto desempenho. que pode ser executada em qualquer ambiente. O gRPC segue amplamente a semântica HTTP sobre HTTP/2. Assim como em C, a proposta do gRPC é que o cliente interaja com o servidor por meio de chamadas de funções simples, ou seja, de interfaces de códigos geradas automaticamente pela própria aplicação do gRPC. Isso significa que você precisa apenas implementar sua lógica de programação, o que facilita muito a adoção do recurso de RPC.

A seguir temos uma tabela comparativa da solução gRPC x REST (XML-RPC):

| Funcionalidade | gRPC | Rest |
|-----------------|----------------------------|------------------------------|
| Protocol | HTTP/2(fast) | HTTP/1.1(slow) |
| Payload | Protobuf (binary/ ,small) | JSON (text, large) |
| API Contract | Strict, required (.proto) | Loose optional (OpenAPI) |
| Code generation | Built-in (protoc) | Third-party tools (Swagger) |
| Security | TSL/SSL | TLS/SSL |
| Streaming | Bidirectional streaming | Client ->server request only |
| Browser support | Limited (require gRPC-web) | Yes |

Fonte: dev.to

Por quê priorizar o uso de gRPC?

- Sistemas multilinguagem de programação: com seu suporte de geração de código nativo para uma ampla gama de linguagens de desenvolvimento, o gRPC é uma boa opção para gerenciar conexões em um ambiente desenvolvido em diferentes linguagens de programação.

³ <https://grpc.io/>

CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA

- *Streaming*: quando a comunicação em tempo real é um requisito, a capacidade do gRPC de gerenciar conexões bidirecionais (HTTP/2) permite que seu sistema envie e receba mensagens em tempo real sem esperar pela resposta do cliente.
- Redes de baixa largura de banda e de baixa potência: o uso de mensagens *protobuf* serializadas, binárias, permite mensagens leves, o que contribui para maior eficiência e velocidade em redes de baixa potência e de largura de banda restritas (especialmente quando comparadas a JSON). A IoT seria um exemplo desse tipo de rede que poderia se beneficiar das APIs gRPC.

O protobuf, por sua vez, é um mecanismo criado e usado pelo Google para serializar dados estruturados, independente de linguagem ou plataforma. Resumidamente, o Protobuf funciona da seguinte maneira:

- Primeiro é definido como se deseja que os dados sejam estruturados em um arquivo de extensão *.proto*;
- Em seguida, esta definição é compilada e o resultado é um código-fonte automaticamente gerado na linguagem desejada. Atualmente as linguagens compatíveis são C++, C#, Go, Java e Python;
- Finalmente, o código-fonte gerado *stub* é utilizado para ler e escrever os dados estruturados por meio da rede.

Neste laboratório, o arquivo *grpcCalc.proto* possui uma sintaxe própria (definida pelo padrão Protobuf de interoperabilidade) para estruturar os dados da comunicação RPC. O próximo passo é gerar os *stubs* cliente e servidor a partir do arquivo *proc*. Neste laboratório, assumindo que o arquivo *.proto* está na mesma pasta que os arquivos *grpcCalc_client.py* e *grpcCalc_servidor.py* use o seguinte comando (na pasta . onde o arquivo *.proto* se encontra):

```
$python -m grpc_tools.protoc --proto_path=. ./grpcCalc.proto --python_out=. --grpc_python_out=.
```

Finalmente, execute inicialmente a aplicação servidor *grpcCalc_server.py* em um terminal e, posteriormente, execute a aplicação cliente *grpcCalc_client.py* e teste a calculadora.

CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA

Atividade 4: Mineração de Criptomoedas usando gRPC (para casa)

Com base nestes exemplos, você precisará construir um protótipo similar a um minerador de criptomoedas no modelo cliente/servidor baseado em RPC. A implementação do minerador deverá ser realizada em Python/gRPC.

Para os alunos de graduação, cada implementação poderá ter seu próprio cliente e servidor, não há exigência (por enquanto) de interoperabilidade entre os diferentes grupos. Porém, para os alunos da pós-graduação, cada implementação poderá ter seu próprio cliente e servidor, mas certifiquem-se de que haja interoperabilidade entre as implementações dos diferentes grupos. Ou seja, o cliente de um grupo poderá chamar o procedimento remoto de outro grupo, e vice-versa.

O servidor deverá ter o seguinte funcionamento:

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

| <i>TransactionID</i> | <i>Challenge</i> | <i>Solution</i> | <i>Winner</i> |
|----------------------|------------------|-----------------|---------------|
| int | int | str | int |

- *TransactionID*: Identificador da transação, representada por um valor inteiro;
- *Challenge*: Valor do desafio criptográfico associado à transação, representado por um número [1..6], onde 1 é o desafio mais fácil. Gere desafios aleatórios ou sequenciais (experimente as diferentes abordagens);
- *Solution*: 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 aos clientes RPC:

**CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA**

| Nome | Parâmetros | Significado |
|-------------------------------|---|---|
| <i>getTransactionID()</i> | <i>none</i> | Retorna o valor atual <int> da transação com desafio ainda pendente de solução. |
| <i>getChallenge()</i> | <i>int transactionID</i> | Se <i>transactionID</i> for válido, retorne o valor do desafio associado a ele. Retorne -1 se o <i>transactionID</i> for inválido. |
| <i>getTransactionStatus()</i> | <i>int transactionID</i> | Se <i>transactionID</i> 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 <i>transactionID</i> for inválida. |
| <i>submitChallenge()</i> | <i>int transactionID, int ClientID, string solution</i> | Submete uma solução (<i>solution</i>) para a função de <i>hashing</i> SHA-1 que resolve o desafio proposto para a referida <i>transactionID</i> . Retorne 1 se a solução for válida, 0 se for inválida, 2 se o desafio já foi solucionado, e -1 se a <i>transactionID</i> for inválida. |
| <i>getWinner()</i> | <i>int transactionID</i> | Retorna o <i>clientID</i> do vencedor da transação <i>transactionID</i> . Retorne 0 se <i>transactionID</i> ainda não tem vencedor e -1 se <i>transactionID</i> for inválida. |
| <i>getSolution()</i> | <i>int transactionID</i> | Retorna uma estrutura de dados (ou uma tupla) com o <i>status</i> , a solução e o desafio associado à <i>transactionID</i> . |

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 servidor RPC e um menu com as seguintes opções:

| Item do Menu | Ação |
|-------------------------|--|
| <i>getTransactionID</i> | Solicita ao servidor a transação corrente (atual) e imprime resultado dessa chamada de função na tela. |
| <i>getChallenge</i> | Lê um valor de <i>transactionID</i> no teclado e solicita o valor do desafio associado a essa transação ao servidor. Imprime esse valor na tela. |

CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA

| | |
|-----------------------------|---|
| <i>getTransactionStatus</i> | Lê um valor de <i>transactionID</i> no teclado e solicita o estado da referida transação para o servidor. Imprime o resultado na tela. |
| <i>getWinner</i> | Lê um valor de <i>transactionID</i> no teclado e solicita o <i>clientID</i> do vencedor ao servidor RPC. Imprime resultado na tela. |
| <i>GetSolution</i> | Lê um valor de <i>transactionID</i> no teclado e solicita a solução associada a essa <i>transactionID</i> . Imprime resultado na tela. |
| <i>Mine</i> | <ol style="list-style-type: none">1. Buscar <i>transactionID</i> atual;2. Buscar a <i>challenge</i> (desafio) associada à <i>transactionID</i> atual;3. Buscar, localmente, uma solução para o desafio proposto – sugere-se usar múltiplas threads para isso!!!!4. Imprimir localmente a solução encontrada;5. Submeter a solução ao servidor e aguardar resultado;6. Imprimir/Decodificar resposta do servidor. |

4. Instruções Gerais

- 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. Se for necessário, o professor reserva-se no direito de ter que subdividir grupos já existentes;
- Data de Entrega: 01/05/2023, por meio da Sala de Aula Virtual da disciplina no *Google Classroom*, na atividade correspondente ao Laboratório III. 1 (uma) submissão por grupo é suficiente;
- Deve-se submeter apenas o link para o repositório virtual da atividade (Github, Bitbucket, Google Colaboratory ou similares) contendo: i) códigos-fonte; ii) instruções para compilação e execução; iii) relatório técnico (.pdf ou README); e iv) vídeo curto (máx 3 min) mostrando uma execução, resultado e análise dos resultados encontrados;
- O relatório técnico deverá conter: a metodologia de implementação, testes e resultados encontrados;

Bom trabalho!