CCF 355 - Sistemas Distribuídos e Paralelos

Trabalho prático - parte 4

Infity Deck

Isabella Ramos (3474), Ricardo Spínola (3471)

22 de julho de 2022

Conteúdo

1 Introdução

O trabalho prático parte 4 consiste na implementação do middleware RMI gRPC no sistema já em desenvolvimento, Infinity Deck.

No gRPC, uma aplicação cliente pode chamar diretamente um método em uma aplicação servidor em uma máquina diferente como se fosse um objeto local, facilitando a criação de aplicativos e serviços distribuídos.

No desenvolvimento da parte do servidor, utilizamos a implementação do gRPC para Python. Já na parte do cliente, utilizamos a implementação do gRPC para o Node.js.

2 Instalação

2.1 Desafios

Para essa parte do trabalho houve complicações na implementação do gRPC junto ao ElectronJS (Framework para interface gráfica). A utilização do gRPC foi executada com êxito em ambientes de testes, mas ao gerarmos o executável da aplicação, tornara problemático o uso do RMI. Problemas em aberto relacionam uma incompatibilidade entre o gRPC e ElectronJS como sugere esta página de reporte no GitHub.

2.2 Utilização

Continua sendo necessário o download da aplicação. A mesma encontra-se disponível nos seguintes links:

- **Linux** → Google Drive ou One Drive
- **Windows** → Google Drive ou One Drive

Para instalação em linux, o arquivo é disponibilizado no formato .deb

- **sudo dpkg -i infinity-deck-rmi.deb**

Como houveram complicações nessa parte, será necessário executar a aplicação direto na pasta de origem.

1. **cd /opt/infinity-deck**
2. **infinity-deck –no-sandbox**

Obs: Caso se tenha um destino de instalação diferente do padrão, deve-se usar o comando **dpkg -L infinity-deck** para verificar onde o pacote foi instalado.

A terceira linha de saída da imagem abaixo mostra o caminho:

```
root@Nemesis:/opt/infinity-deck# dpkg -L infinity-deck
/.
/opt
/opt/infinity-deck
/opt/infinity-deck/chrome-sandbox
/opt/infinity-deck/chrome_100_percent.pak
/opt/infinity-deck/chrome_200_percent.pak
/opt/infinity-deck/chrome_crashpad_handler
/opt/infinity-deck/controller.proto
```

Figura 1: Resultado do comando dpkg -L infinity-deck

2.3 Utilização - Servidor

Para o servidor, mantém-se os mesmos procedimentos. Os arquivos são encontrados dentro da pasta Server.

- **python3 init.py** *Inicialização do banco (reset)*
- **python3 server.py** *Inicialização do servidor em 127.0.0.1 - 23123*

3 Implementação

O gRPC possui alguns tipos de implementação, o que utilizamos foi o gRPC unário: um gRPC simples que funciona como uma chamada de função normal. Ele envia uma única solicitação declarada no arquivo protobuff (*.proto*) para o servidor e recebe de volta uma única resposta do servidor.

3.1 Implementação servidor em Python

Para a implementação no servidor, utilizamos a referência [2] para o desenvolvimento em Python. Para instalar as dependências necessárias utilizamos o comando da Figura 2.

```
pip install grpcio grpcio-tools
```

Figura 2: Comando para instalação das dependências em Python

Já na implementação de serviços do gRPC foi necessário a definição do arquivo protobuff, onde o mesmo compreende a declaração do serviço que é usado para gerar subs, que são no nosso caso, *controller_pb2.py* e *controller_pb2_grpc.py*. Além disso, foi necessário um arquivo de servidor gRPC que atende as solicitações do cliente.

Como podemos ver na Figura 3, foi declarado um serviço Controller, que é responsável por especificar os métodos que podem ser chamados remotamente.

O buffer dos dados de protocolo é estruturado em *messages*, onde cada *message* é um pequeno registro lógico de informações contendo uma série de pares nome-valor chamados de *fields*. Na nossa implementação, o método executeOperation() recebe uma entrada do tipo *Message* e retorna um *MessageResponse* sendo que os dois últimos foram declarados abaixo da declaração do serviço.

Além disso, utilizamos um método para verificar a disponibilidade do servidor. O método *serverAlive* não precisa de entradas ou saídas para seu funcionamento, baseia-se em *timeout*, por isso o uso do *message Empty*.

```
syntax = "proto3";

package controller;

service Controller{

    rpc executeOperation(Message) returns (MessageResponse) {}
    rpc serverAlive(Empty) returns (Empty) {}

}

message Empty{
}

message Message{
    string message = 1;
}

message MessageResponse{
    string message = 1;
}
```

Figura 3: controller.proto

Após a criação do arquivo *.proto* deve-se gerar os stubs (*controller_pb2.py* e *controller_pb2_grpc.py*). Para isso devemos executar o código da Figura 4. Após esse passo, já é possível fazer a implementação do servidor.

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

Figura 4: Comando para gerar os stubs

A implementação do servidor foi feita no arquivo *server.py*. Nele devemos importar o pacote grpc , os arquivos gerados pelo *controller.proto* juntamente com a classe *Controller* do arquivo contoller.py usada para realizar operações no banco.

A primeira parte da implementação do gRPC no backend foi feito na classe *ControllerService* que está na figura 5. Nela é passado como parâmetro uma das classes do arquivo *controller_pb2_grpc.py*, arquivo este que foi gerado pelo protobuff.

ControllerService deve definir o corpo dos métodos utilizados remotamente.

```
class ControllerService(pb2_grpc.ControllerServicer):

    def __init__(self, *args, **kwargs):
        self.controller = Controller()
        pass

    def executeOperation(self, request, context):

        message = request.message
        resp = self.controller.executeOperation(message)
        result = {'message': resp}
        return pb2.MessageResponse(**result)

    def serverAlive(self,request,context):
        return pb2.Empty()
```

Figura 5: Classe implementada para a utilização do gRPC

```
def serve():
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))

    pb2_grpc.add_ControllerServicer_to_server(ControllerService(), server)

    print("\n\033[1;35m[*] On: {} - {}\033[0;0m\n".format(HOST,PORT))
    server.add_insecure_port('[::]:{}'.format(PORT))
    server.start()
    server.wait_for_termination()
```

Figura 6: Função serve() implementada no serve.py

3.2 Implementação cliente em Node.js

Para a implementação do cliente, utilizamos a referência [1] para o desenvolvimento em Node.js.

Foi utilizado o mesmo código da figura 3, também chamado de *controller.proto*, mas desta vez dentro da pasta da interface.

O cliente é instanciado no arquivo *index.js*. Nele é importado o pacote do gRPC, o pacote do protoloader do gRPC e o arquivo do protobuff.

A variável *client_grpc* estabelece conexão com o servidor ao instanciar a classe *Controller*.

```
const packageDefinition = protoLoader.loadSync(PROTO_PATH,options);
const Controller = grpc.loadPackageDefinition(packageDefinition).controller.Controller;

let PORT = 23123
let HOST = '127.0.0.1'
let serverLive = true

let client_grpc = new Controller(`${HOST}:${PORT}`,
    grpc.credentials.createInsecure())
```

Figura 7: Arquivo index.js

```
function sendMessage(request){
    return new Promise(resolve=>{
        client_grpc.executeOperation({message:request}, (error, response) => {
                if(error && error.code === 14){
                  // Server desconectado
                  resolve({status:false,message:"Servidor desconectado !"})
                }
                else {
                  resolve(JSON.parse(response.message))
                }
            });
    })
}
```

Figura 8: Função sendMessage

A figura 8 define como a aplicação cliente comunica com o servidor. Através da função *sendMessage* a interface consegue enviar mensagens para o servidor e esperar por uma resposta com o uso de Promises

Mesmo com o uso do gRPC, a comunicação por IPC é presente, e destina-se a comunicar interface gráfica com aplicação cliente.

Para verificarmos a disponibilidade do servidor por parte da interface, a mesma dispara em intervalos de tempo a execução do método *serverAlive*.

É estabelecido um intervalo de 2s entre cada chamada, e um *timeout* também de 2s. Caso este seja ultrapassado, a aplicação torna-se indisponível até o servidor estar online novamente.

```
function serverAlive(){
    let timeout = new Date( Date.now() + 2000)
    client_grpc.serverAlive({},{deadline:timeout},(error,response) => {
        if(error && (error.code===14 || error.code===4)){ // Código para indisponibilidade
          serverLive=false
        }
        else{serverLive=true}
    })
}

serverAlive()
setInterval(() => {
  serverAlive()
}, 2000);
```

Figura 9: Função serverAlive

4 Mudanças

Removemos a funcionalidade de se alterar o ip e porta da aplicação. Essa decisão foi tomada por não encontrarmos solução por parte do gRPC em nodeJS para tal ação.

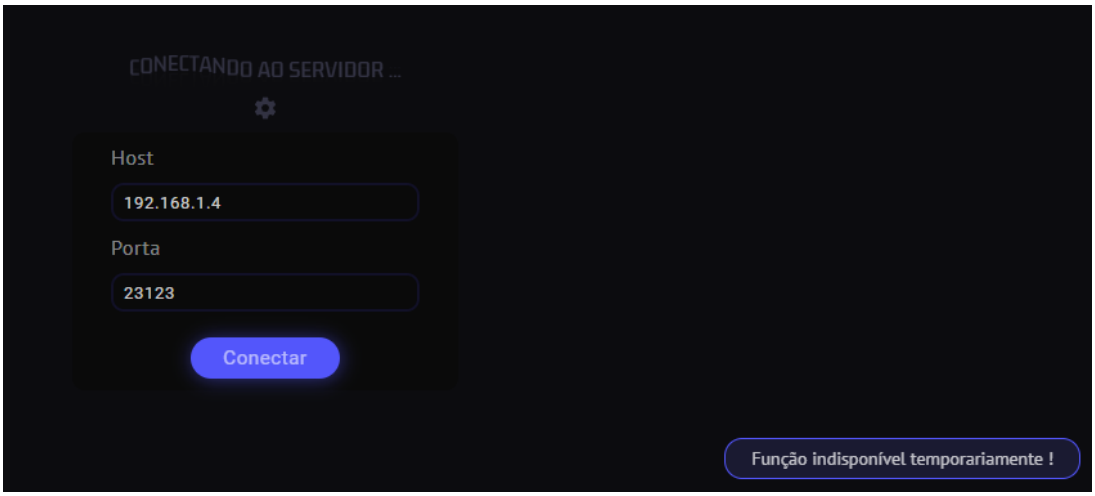

Figura 10: Mudança feita na aplicação

5 Conclusão

A utilização do middleware RMI gRPC possibilitou abstrair problemas encontrados quando se implementava sockets diretamente. Um deles seria a tratativa para extensos conjuntos de dados, que para nosso sistema, utilizávamos flags de início e fim.

Ademais, com o uso do RMI, minimizamos a codificação para estabelecermos uma comunicação concorrente. O gRPC define em sua instância no servidor, uma *thread* por execução de método.

Visto que a interface não sofreu alterações, ocultamos características da mesma que fora mostrada na parte 3 do trabalho prático.

Referências

[1] Thiago S. Adriano. *Introdução ao gRPC com Node.js*. en. Dez. de 2019. URL: https://medium.com/xp-inc/introdu%C3%A7%C3%A3o-ao-grpc-com-node-js-98f6a4ede11 (acedido em 17/07/2022).

[2] *Implementing gRPC In Python: A Step-by-step Guide*. URL: https://www.velotio.com/engineering-blog/grpc-implementation-using-python (acedido em 17/07/2022).