UNIVERSIDADE FEDERAL DE VIÇOSA - *CAMPUS* FLORESTAL
Bacharelado em Ciência da Computação

Adriano Marques Martins 02640
Eduardo Vinicius Bittencourt Esquivel 03498

TRABALHO PRÁTICO

Parte 04

FLORESTAL
2021

Adriano Marques Martins 02640
Eduardo Vinicius Bittencourt Esquivel 03498

TRABALHO PRÁTICO

Parte 04

Descrição das decisões de projeto na implementação do trabalho prático parte 4 usando o gRPC.

Disciplina: CCF 355 - Sistemas Distribuídos e Paralelos

Prof(a).: Thais Regina de M. B. Silva

Florestal
2021

Sumário

Linguagens, Bibliotecas e Padrões

A dupla optou pela troca do banco de dados, e desta vez, os dados estão sendo persistidos em um banco de dados SQLite que está sendo enviado em conjunto com o projeto, bem como uma pasta com o código que foi utilizado para a geração do banco.

As demais decisões continuaram inalteradas, como a linguagem Python, o ambiente de desenvolvimento PyCharm, o uso do Git e GitHub para controle de versões, o padrão MVC para modular o projeto e a biblioteca Tkinter para a interface gráfica.

A dupla escolheu o sistema gRPC para que o cliente chamasse diretamente um método do lado do servidor como se fosse uma função local.

O projeto conta com um arquivo denominado "*requirements*.txt" que pode ser chamado para que sejam instaladas todas as bibliotecas necessárias para que o projeto e o sistema gRPC funcione corretamente. Recomendamos o uso de um ambiente virtual.

```
python -m venv venv
.\venv\Scripts\activate
pip install -r requirements.txt
```

Código 01. Criação do ambiente virtual, ativação do ambiente e instalação dos pacotes necessários.

Arquivo .proto

Por padrão, o gRPC usa Buffers de Protocolo como a IDL (Interface Definition Language) para serializar dados estruturados até mesmo JSON, uma vez que dada a terceira parte do trabalho, muitas funções do banco de dados retornavam dicionários de dados, a dupla optou por manter estes dicionários e enviá-los como resposta para o cliente.

O arquivo .proto tem, portanto, a IDL que foi compilada para o Python através do comando:

```
python -m grpc_tools.protoc --proto_path=. .\proto\message.proto
--python_out=. --grpc_python_out=.
```

Código 02. Compilação do arquivo .proto para a linguagem Python.

O gRPC é baseado na ideia de definir serviço, especificando quais métodos podem ser acessados remotamente, com seus parâmetros e o tipo de retorno. Desta forma, como pode ser visto no código abaixo, definimos que o cliente poderá chamar oito funções do servidor, sendo elas respectivamente: Login, criar conta, ver o álbum, comprar um pacote de figurinhas, vender uma figurinha, ofertar uma troca, ver as trocas possíveis e trocar figurinhas.

```
service Message{
    rpc Login(MessageClient) returns (LoginResponse) {}
    rpc Create(MessageClient) returns (Response) {}
    rpc Album(MessageClient) returns (AlbumResponse) {}
    rpc Buy(MessageClient) returns (AlbumResponse) {}
    rpc Sell(MessageClient) returns (SellResponse) {}
    rpc CreateTrade(MessageClient) returns (Response) {}
    rpc ListTrade(MessageClient) returns (ListTradeResponse) {}
    rpc Trade(MessageClient) returns (Response) {}
}
```

Código 03. Funções Remotas e seus respectivos parâmetros e retornos.

Como pode ser notado, para todas as funções do servidor são passados os mesmo parâmetros e essa foi uma decisão da dupla. Como quase todas as funções necessitavam de um ou dois parâmetros do mesmo tipo, não faria sentido criar várias mensagens distintas. Poderíamos simplificar e colocar apenas 3 parâmetros, mas optamos por manter os nomes para manutenção da legibilidade ao invés de nomes genéricos.

```
message MessageClient{
  optional int32 idUser = 1;
  optional int32 idFigure = 2;
  optional int32 offer = 3;
  optional int32 taking = 4;
  optional string name = 5;
  optional string password = 6;
  optional int32 idTrade = 7;
}
```

Código 04. Proto dos Parâmetros

Já o servidor, algumas funções retornavam apenas uma resposta simples de conformação como criar conta, anunciar uma troca e a função trocar, e nestes casos, usamos uma mensagem simples que contém um valor booleano.

```
message Response{
  bool response = 1;
}
```

Código 05. Retorno simples

O retorno da função de login (LoginResponse) necessitaria dos dados do usuário e uma figura que poderia vir ou não com o login diário. Para isso o objeto foi estruturado de forma a conter um objeto usuário, um booleano se ocorreu tudo bem e um objeto contendo a figura.

```
message Figure{
```

```
  int32 idFigure = 1;
  string name = 2;
  string rarity = 3;
  optional string path = 4;
  optional int32 quantity = 5;
  optional int32 idUser = 6;
}
message User{
  int32 idUser = 1;
  string name = 2;
  float balance = 3;
  string password = 4;
  bool showcard = 5;
}

message LoginResponse{
  bool response = 1;
  User user = 2;
  optional Figure figure = 3;
}
```

Código 06. Retorno do login e seus objetos: Figura e Usuário.

Tanto para a função de ver álbum quanto a função comprar retornam o mesmo objeto que contém um valor booleano se ocorreu tudo certo, se o álbum foi completado, o saldo, uma figurinha especial caso o álbum tenha sido completado e a lista de figurinhas, que dependendo do contexto pode ser todas do álbum ou a lista de figurinhas adquiridas através da compra.

```
message AlbumResponse{
  bool response = 1;
  optional bool complete = 2;
  optional int32 idUser = 3;
  optional Figure special = 4;
  repeated Figure figures = 5;
  optional float balance = 6;
}
```

Código 07. Retorno do compar e ver álbum.

Ao vender uma figurinha, a função de Sell deve retornar um valor booleano se ocorreu tudo certo, o saldo e o nome da figurinha vendida.

```
message SellResponse{
  bool response = 1;
  float price = 2;
  string name = 3;
```

```
}
```

Código 08. Retorno da venda de figurinhas.

Por fim, a função de ver trocas, lista todas as trocas possíveis. Portanto a função deve retornar um valor booleano se a operação foi bem sucedida e uma lista de trocas, que contém: O nome de quem está anunciando a troca, o identificador da troca e os dados da figurinha anunciada e da figurinha desejada.

```
message Trade{
  string name = 1;
  int32  idTrade = 2;
  int32 offerID = 3;
  string offerName = 4;
  string offerRarity = 5;
  int32 takingID = 6;
  string takingName = 7;
  string takingRarity = 8;
}

message ListTradeResponse{
  bool response = 1;
  repeated Trade list = 2;
}
```

Código 09. Retorno da lista de trocas e seu objeto de Troca.

Servidor

Como desta vez não foi necessário o uso de uma função controladora do lado do servidor para chamar a função correta para a mensagem do cliente, o servidor ficou muito mais simplificado e limpo, pois com o uso do sistema gRPC muitas coisas de definição ficaram mais sucintas.

```
SERVER..
Login -> Name: login  - Password:  login
Response <-  {'response': False, 'user': None, 'figure': None}
Login -> Name: teste  - Password:  teste
Response <-  {'response': True, 'user': {'idUser': 1, 'name': 'teste', 'balance': 8350.0, 'password': 'teste', 'showcard': 1}, 'figure': {'idFigure': 10, 'rari
Listar Trocas ->
response <- [{'name': 'teste', 'idTrade': 1, 'offerID': 1, 'offerName': 'Monkey D. Luffy', 'offerRarity': 'RARA', 'takingID': 3, 'takingName': 'Roronoa Zoro',
Album -> IdUser: 1
Response <-  {'response': True, 'complete': 0, 'special': None, 'figures': [{'idFigure': 1, 'name': 'Monkey D. Luffy', 'rarity': 'RARA', 'path': 'Monkey D Luff
Login -> Name: teste  - Password:  teste
Response <-  {'response': True, 'user': {'idUser': 1, 'name': 'teste', 'balance': 8350.0, 'password': 'teste', 'showcard': 0}, 'figure': None}
Album -> IdUser: 1
Response <-  {'response': True, 'complete': 0, 'special': None, 'figures': [{'idFigure': 1, 'name': 'Monkey D. Luffy', 'rarity': 'RARA', 'path': 'Monkey D Luff
```

Código 10. Troca de mensagem entre o cliente e servidor

Cliente

Diferentemente do servidor, o cliente teve um aumento de código. Entretanto, o código acrescentado não foi nada complexo, pois eram apenas funções locais que criavam a mensagem usada como parâmetro e a chamada para a função remota do servidor.

```
class Client(object):

   def __init__(self):
       self.host = 'localhost'
       self.server_port = 50051
       # Instanciando o Canal
       self.channel = grpc.insecure_channel('{}:{}'.format(self.host,
self.server_port))
       # Ligando o cliente ao Servidor
       self.stub = pb2_grpc.MessageStub(self.channel)

   def login(self, name, password):
       message = pb2.MessageClient(name=name, password=password)
       return self.stub.Login(message)

 { ... }
```

Código 11. Código cortado do cliente que faz a invocação remota do método login do servidor.

Interfaces

Neste projeto, toda a interface gráfica foi implementada, com todas as funcionalidades, como login, criar conta, ver álbum, comprar figurinha, vender figurinha, e o menu de trocas que contém a opção de anunciar trocar, ver as trocas possíveis e realizar troca. A interface gráfica, desta vez, está acessível para outros tamanhos de telas.

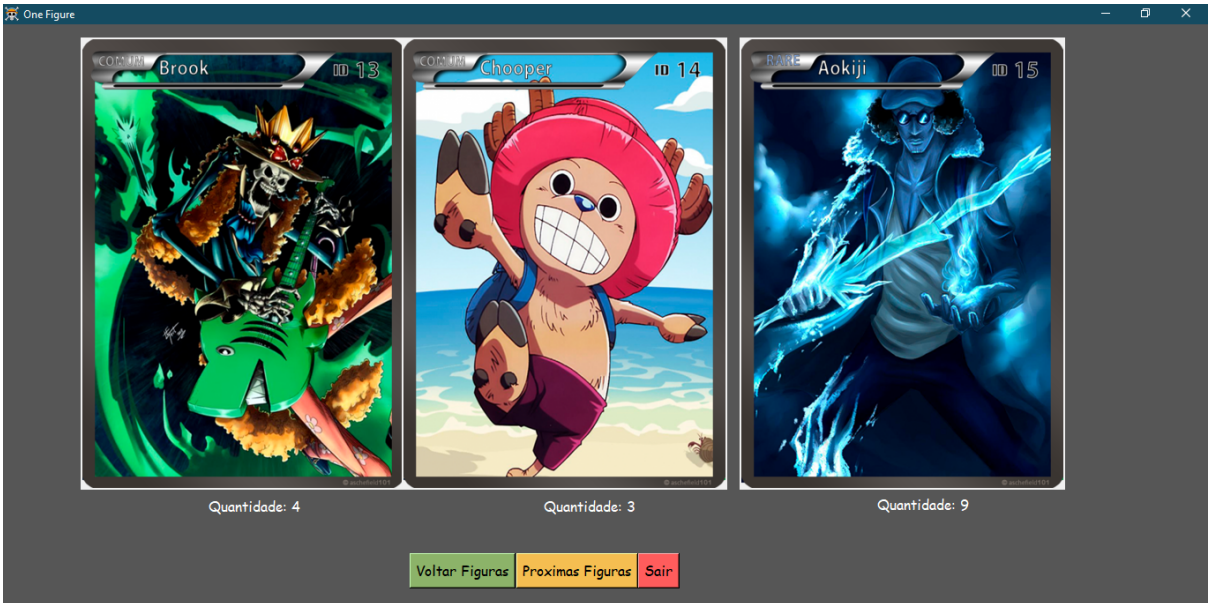

Figura 01. Exemplo da interface gráfica mostrando o álbum de figurinhas

A interface do terminal também continua inalterada como pode ser vista na figura 2.

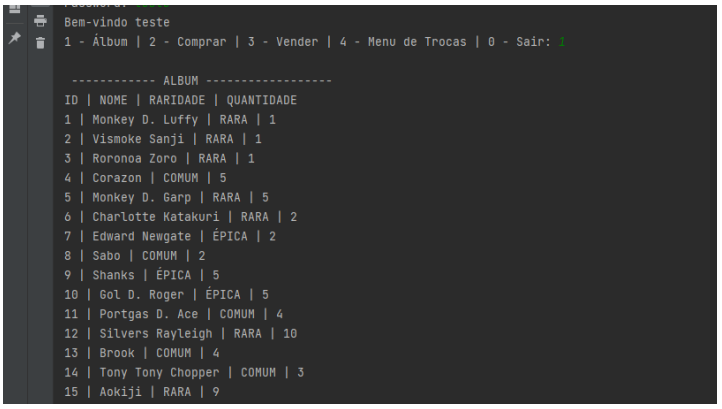

Figura 02. Exemplo da interface do terminal mostrando o álbum de figurinhas.

O arquivo **“server.py”** na raiz do projeto inicia o servidor, o arquivo **“client.py”** inicia o cliente via terminal, enquanto o arquivo **“cliente-interface.py”** inicia o cliente com interface.

Conclusão

A dupla teve dificuldades ao usar o sistema gRPC, principalmente na formatação da mensagem. Enquanto com o uso de sockets a dupla foi livre para formatar como bem entendesse a mensagem, ao usar o gRPC a dupla teve que seguir o padrão definido na sintaxe da IDL.

Como as consultas do banco de dados estavam retornando dicionários de dados, que antes eram usados para se criar JSONs, agora acabavam muitas vezes gerando erros ao enviar a mensagem e a atribuição apresentava várias inconsistências com o nome dos atributos dos objetos.

Apesar do gRPC tirar um pouco a liberdade, ele trouxe vários benefícios, como dito anteriormente, a clareza de código. Anteriormente, com o uso de sockets, teríamos que tratar de muitas questões quanto ao formato da mensagem e desta vez, com um padrão definido pela IDL muitos destes tratamentos puderam ser eliminados.

Outro benefício do gRPC foi eliminar o tratamento quanto ao tamanho da mensagem, pois com o uso de sockets, tínhamos que definir o tamanho da mensagem que uma hora poderia ocorrer algum truncamento. Mas nesta nova implementação, isso foi abstraído do programador, o que na nossa opinião, ajudou em muitas coisas.