Trabalho Prático - Parte 4
Sistemas Distribuídos e Paralelos

Dener Vieira Ribeiro (3872)[1] **Germano Barcelos dos Santos (3873)**[1]

[1]Instituto de Ciências Exatas e Tecnológicas, Campus UFV-Florestal
Universidade Federal de Viçosa

1. Introdução

Este trabalho consiste na quarta etapa da implementação de um sistema distribuído de troca de figurinhas do álbum da copa do mundo. Nas etapas anteriores foram criados os requisitos funcionais, diagrama de casos de uso, e os modelos físicos, de arquitetura e fundamentais, e a primeira implementação que utilizava a API de Sockets como forma de troca de mensagens entre processos. Nessa parte, foi desenvolvida a segunda implementação do projeto e, para isso, foi utilizado o gRPC para substituir o meio de troca de mensagens utilizado anteriormente.

O gRPC é uma biblioteca que permite a invocação remota de métodos em processos diferentes. Para isso, é necessário definir um serviço, especificar os métodos que podem ser chamados remotamente com seus parâmetros e tipos de retorno. As mensagens a serem trocadas foram definidas utilizando Protocol Buffers (protobuf), que é uma linguagem neutra para serialização de estruturas de dados.

2. Desenvolvimento

O sistema distribuído criado é composto por dois processos principais, o servidor e o cliente. O processo servidor é responsável por implementar os métodos que serão invocados pelos usuários. O cliente faz a invocação remota sempre que necessário. A seguir são apresentadas as descrições das principais funcionalidades de cada processo. Além disso, são discutidas as modelagens do banco de dados, as implementações chamada dos métodos e das threads utilizadas.

2.1. Modelagem do Sistema

Primeiramente, é preciso discutir como o sistema foi modelado para entender o contexto da comunicação e como a troca de mensagens é realizada. O sistema utiliza o banco de dados relacional *SQLite* com o auxílio do *framework SQLAlchemy* que é uma ferramenta que facilita as consultas e a criação das tabelas a partir de definição declarativa. Portanto, criamos classes que representam diretamente uma tabela, com suas relações, o diagrama relacional pode ser visto na figura 1. A tabela *users* guarda os dados dos usuários e a tabela *stickers* guarda todos as figurinhas.

É importante ressaltar as duas relações que foram feitas. A tabela *list_stickers* é criada para armazenar as figurinhas de um usuário, e por isso configura-se uma relação $M : N$, ou seja um usuário pode ter M figurinhas e uma figurinha pode ter sido sorteada para N usuários. A tabela *trade* é a responsável por armazenar todas as trocas de um usuário u_1 para outro u_2, que pode ser aceita ou recusada. Quando a troca é solicitada, seu *status* é pendente. A tabela *trade_stickers* é necessária para armazenar quais as trocas serão feitas no troca t_i, pois o usuário u_1 pode trocar N por M figurinhas. Nesse caso,

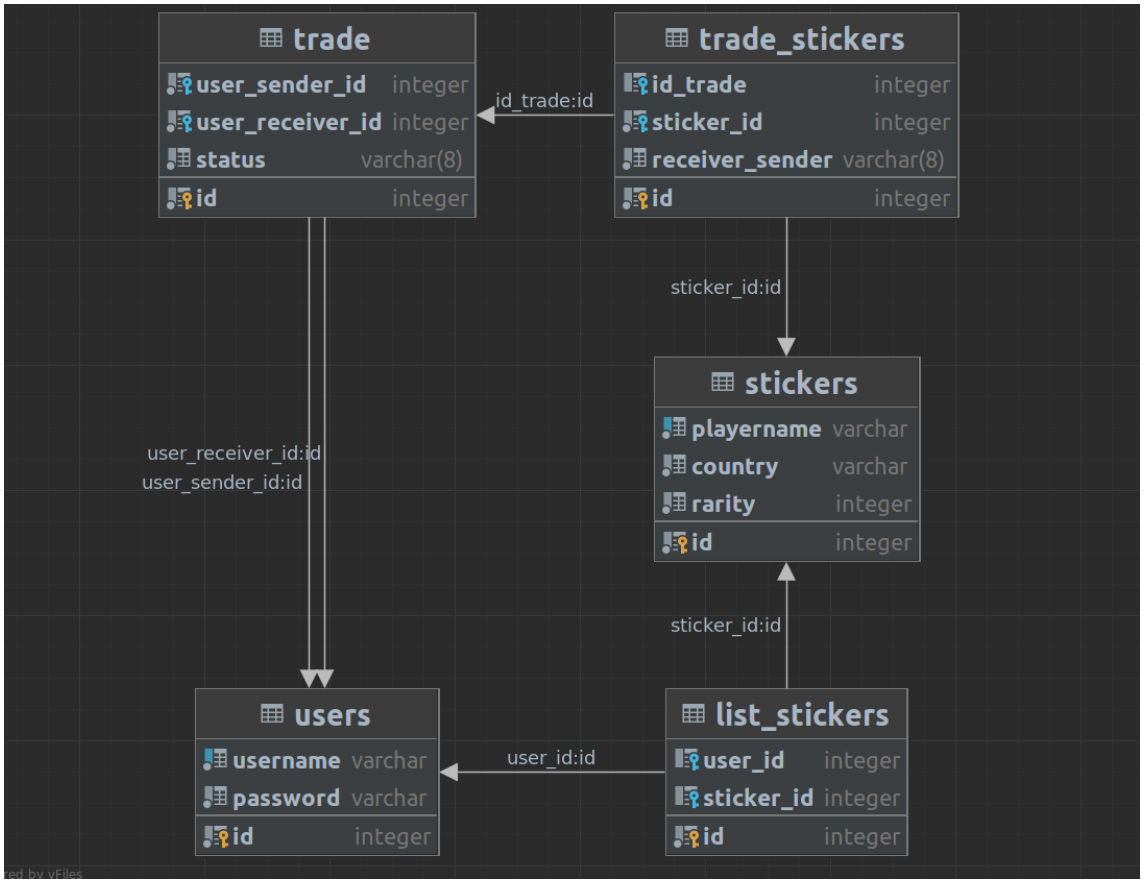

Figura 1. Digrama Relacional

identificamos quem é vai receber e quem vai enviar por meio de um campo chamado *receiver_sender* que pode ser *receiver* para destinatário e *remetente* para remetente.

Selecionamos 88 figurinhas, o time titular de cada seleção que achamos que chegará às quartas de finais: Holanda, Bélgica, Argentina, Espanha, Portugal, Brasil, França e Inglaterra. Cada vez que um usuário é criado sorteamos k figurinhas para o mesmo, com peso w, de acordo com a raridade $r \in \{1, 2, 3\}$, sendo 3 a maior raridade. Para essa finalidade, utilizamos a função *random.choices* do *python*, que cria um vetor com k posições. Após a criação desse vetor, executamos k consultas ao banco de dados pela raridade k_i ordenando randomicamente e selecionando o primeiro da lista. Assim, permitimos ter duplicados e uma distribuição justa de figurinhas para cada usuário.

Além dessa modelagem por classes, implementamos o padrão *repository* que ficará responsável por executar todas as consultas e fazer as atualizações necessárias do banco de dados. Dessa maneira toda a persistência fica desacoplada e responsável por somente uma tarefa.

2.2. Processo Cliente

O processo cliente fornece ao usuário uma interface gráfica onde é possível usar de forma simples as funcionalidades de visualizar figurinhas, solicitar uma troca de figurinhas com algum outro usuário, ver as solicitações recebidas, e sair do sistema.

Cada requisição feita pelo usuário é transformada em uma chamada de método no

servidor, em seguida, o processo cliente recebe uma resposta do servidor com os resultados da operação.

2.3. Processo Servidor

O processo servidor, inicialmente cria uma variável do tipo servidor do gRPC com uma pool de threads, que será utilizado para trocar mensagens com algum processo cliente, de maneira paralela. Após isso, o processo cria o um canal especificando a porta, na qual, as mensagens serão recebidas/enviadas.

2.4. gRPC

Os arquivos *protobuf* são essenciais para o sistema, pois são representações de objetos que na versão com socket era feito com JSON(JavaScript Object Notation). Logo o gRPC, além de criar classes específicas para o desenvolvimento do projeto, utiliza essa representação para transferir dados entre os serviços clientes/servidores.

Após a definição dos protobufs, o gRPC gera automaticamente classes que representam a interface de serviço, ou seja, são espelhos para os procedimentos que definimos nos arquivos acima. Assim, é possível utilizar essas classes para implementar a comunicação entre o servidor e o cliente.

Para o cliente enviar uma requisição e receber uma resposta do servidor é necessário que o cliente se conecte ao canal aberto para envio e o recebimento de mensagens por parte do servidor e partir disso pode usar a classe stub que é a responsável por enviar e receber as mensagens.

2.5. Comunicação

A comunicação é feita através dos comandos requisição-resposta utilizando a ferramenta gRPC. Definimos 3 serviços, consequentemente 3 arquivos do tipo *protobuf*:

1. Sticker Service
2. Trade Service
3. User Service

Os arquivos estão descritos nos algoritmos .1, .3, .2. Nesses arquivos é possível identificar semelhanças: definição de um serviço com os procedimentos remotos, a especificação de cada mensagem que é recebida/enviada pelo servidor. Comparando com o sistema desenvolvido com Socket, as classes de comandos fazem o papel das *messages*. Por isso, cada classe comando pode ser substituída pelas *messages*. Cada mensagem possui um conjunto de atributos, definindo o conteúdo da mesma. Os procedimentos remotos adotados nesse trabalho, serão todos unários, para facilitar a implementação, para enviar uma lista de uma estrutura é declarado um atributo com modificador repeated.

No serviço *User Service*, há a especificação de 2 procedimentos remotos: login e *register*. O primeiro procedimento recebe o LoginRequest, o nome do usuário e uma senha e retorna uma resposta que contém o id do usuário, se não encontrar o nome do usuário, assumimos que há um erro e assim levantamos uma exceção. O segundo procedimento recebe o CreateRequest, o nome de usuário e uma senha e retorna o status da operação do tipo booleano.

No protobuf .2 temos a especificação do serviço de listar as figurinhas de um usuário específico, portanto o procedimento remoto recebe uma ListStickerRequest, uma

```
service UserService {
    rpc login (LoginRequest) returns (LoginResponse) {}
    rpc register (CreateRequest) returns (CreateResponse) {}
}
message LoginRequest {
    string username = 1;
    string password = 2;
}
message LoginResponse {
    int32 user_id = 1;
}
message CreateRequest {
    string username = 1;
    string password = 2;
}
message CreateResponse {
    bool status = 1;
}
```

Algoritmo .1: Protobuf do *User Service*

```
service StickerService {
    rpc list_stickers (ListStickersRequest) returns
    ↪ (ListStickerResponse) {}
}

message ListStickerResponse {
    message Sticker {
        string playername = 1;
        string country = 2;
        int32 rarity = 3;
        int32 id = 4;
    }

    repeated Sticker sticker = 1;
}

message ListStickersRequest {
    string username = 1;
}
```

Algoritmo .2: Protobuf do *Sticker Service*

mensagem composta pelo nome do usuário e envia um com *ListStickerResponse*, que possui uma lista de figurinhas, definida por nome do jogador, país, raridade e o id correspondente ao banco de dados.

O protobuf .3 é o mais complexo e o essencial para o sistema proposto. O serviço possui 3 procedimentos. O primeiro é o *request_trade* que recebe uma mensagem contendo uma lista de identificadores de figurinhas a enviar e outra lista respectiva às figurinhas que o usuário deseja receber, o nome de usuário que realizou a proposta e o nome de usuário que irá aceitar ou recusar a proposta, e envia uma mensagem contendo um status que especifica se a troca foi cadastrada corretamente ou se ocorreu um erro; O segundo método é o *answer_trade* que recebe uma mensagem que contêm a resposta do usuário: aceita ou recusada, e o id respectivo à troca e retorna um status contendo o status da transação; Já o último procedimento (*get_trades*) é a especificação de buscar todas as trocas pendentes de um usuário, portanto recebe um nome de usuário e retorna as trocas pendentes que foram solicitadas a esse usuário. Cada troca é composta por uma lista de figurinhas que irá receber e a lista que será enviada, o nome de usuário que fez a solicitação e o id respectivo à troca.

2.6. Modificações e Detalhes de Implementação

Durante a implementação do sistema com Sockets, foi necessário fazer algumas implementações mais baixo nível como tratar o tamanho do buffer de leitura com 4096 bytes, definir um caractere como final de mensagem. Além disso, era necessário conferir quando havia um envio de mensagem, qual era o tipo de requisição feita para o servidor e somente depois de todo esse processo, a mensagem era analisada e as operações respectivas ao tipo da requisição eram executadas. Utilizando o gRPC nenhum processamento de mais baixo nível foi realizado, por conta dessa ferramenta ser uma middleware, ou seja, uma camada de software que abstrai toda a implementação de troca de mensagens. Cria-se portanto, serviços que podem ser utilizados como interfaces facilitando a implementação do sistema distribuído.

Contudo a regra de negócio do sistema desenvolvido com Socket foi mantida, como a classe de *TradeStickersService*. Essa classe é responsável somente por realizar operações relacionadas a troca de figurinhas; dessa maneira, a troca é feita de forma independente de socket ou middleware. Além dessa classe, o método de adicionar figurinhas ao usuário quando é criado, também não mudou e continua pertencendo à classe *StickersPack*.

As outras classes de serviço são respectivas à implementação da interface de serviço definida pelo gRPC. É necessário implementar todas as funcionalidades que serão invocadas pelos clientes, para que as operações sejam realizadas dentro do servidor e retornem os objetos solicitados pelos clientes.

Logo, cada resposta dos serviços definidos nos *protobufs* foram implementados de forma a preencher o conteúdo das mensagens com as informações do banco de dados.

2.7. Threads

Além de o sistema usar uma representação externa de dados, suporta múltiplos clientes trocando mensagens com o servidor, através de *threads*. O próprio servidor do gRPC faz o controle de acesso utilizando um *pool* de threads.

```
service TradeService {
    rpc request_trade (TradeRequest) returns (TradeResponse) {}
    rpc answer_trade (AnswerTradeRequest) returns
    ↪ (AnswerTradeResponse) {}
    rpc get_trades (GetTradesRequest) returns (GetTradesResponse)
    ↪ {}
}
message TradeRequest {
    repeated int32 my_stickers = 1;
    repeated int32 other_stickers = 2;
    string my_username = 3;
    string other_username = 4;
}
message TradeResponse {
    bool status = 1;
}
message AnswerTradeRequest {
    bool accept = 1;
    int32 trade_id = 2;
}
message AnswerTradeResponse {
    bool status = 1;
}
message GetTradesRequest {
    string username = 1;
}
message GetTradesResponse {
    message Trade {
        message Sticker {
            string playername = 1;
            string country = 2;
            int32 rarity = 3;
            int32 id = 4;
        }
        enum Status {
            PENDENT = 0;
            ACCEPTED = 1;
            RECUSED = 2;
        }
        repeated Sticker to_send = 1;
        repeated Sticker to_receive = 2;
        string username = 3;
        Status status = 4;
        int32 trade_id = 5;
    }
    repeated Trade trades = 1;
}
```

Algoritmo .3: Protobuf do *Trade Service*

2.8. Interface Gráfica

A interface gráfica foi criada para que seja intuitiva, fácil de utilizar, e que seja capaz de fornecer ao usuário a possibilidade de realizar todas as operações solicitadas nesse trabalho. Foram feitas cinco telas. A tela de entrada tem dois campos que devem ser preenchidos pelo usuário com nome e senha. A de cadastro, contém 3 campos, um de nome e dois de senha, sendo que o segundo pede a confirmação da senha para a realização do cadastro. A página principal contém uma lista com todas as figurinhas do usuário atualmente utilizador do sistema. A tela de trocar contém duas colunas, uma com as figurinhas do usuário atual e outra com as figurinhas de outro usuário que pode ser buscado no sistema através de um campo de busca. Por último, a tela de visualizar solicitações de trocas, apresenta uma solicitação de troca e permite que o usuário navegue pelas solicitações pelos botões de anterior e próximo. As imagens em 2 apresentam todas as telas descritas.

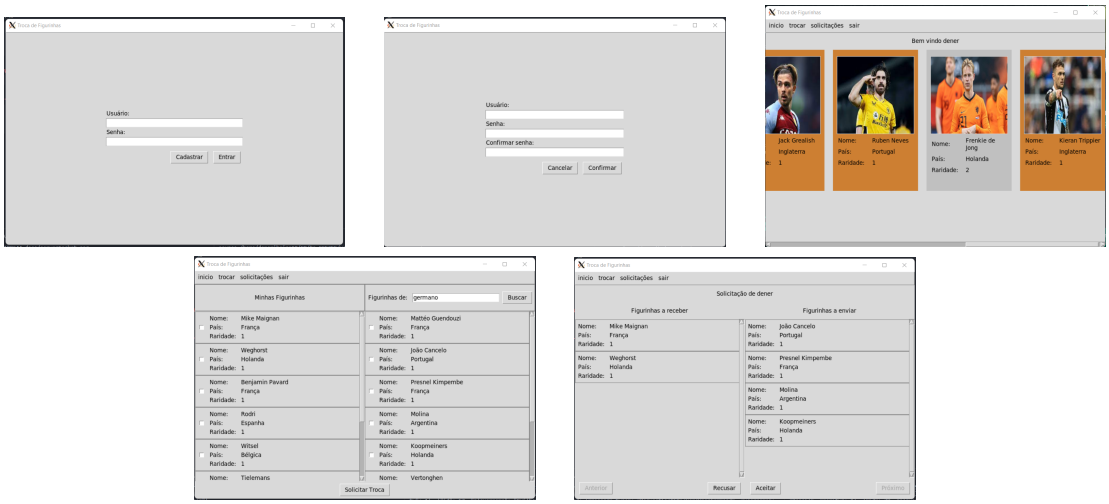

Figura 2. Telas do sisitema

3. Resultados

Com o objetivo de testar o sistema, foram feitos múltiplos testes e, em todos, o resultado foi o esperado. As principais verificações foram de validação de cadastro de usuário, onde não pode haver usuários repetidos. O login deve autenticar um usuário se, e somente se, seu nome e senha estão corretos. E, uma troca pode ser realizada apenas se ambos os usuários possuem as figurinhas presentes no âmbito da troca.

A nova implementação utilizando um middleware, facilita o desenvolvimento do sistema visto que não é mais necessário definir as mensagens a serem trocadas manualmente, nem há preocupação com os detalhes de implementação. Por outro lado, o gRPC gera as classes de forma dinâmica e isso gera uma grande dificuldade ao programador pois os editores não conseguem aplicar análises e correções automáticas no código. Além disso, alguns erros de sintaxe não geram excessões no gRPC, dificultando ainda mais o processo de debugar.

4. Conclusão

Em virtude do que foi apresentado, foi possível perceber que um sistema distribuído que utiliza um middleware apresenta vantagens muito grandes quando comparado à implementação da comunicação por meio de troca de mensagens. Além disso, é possível fazer transmissão de objetos e invocação remota de métodos, e assim, o sistema pode ser visto como um conjunto de objetos.

Ademais, o gRPC transmite informações sobre excessões entre servidor e cliente. Ou seja, mesmo que um método falhe sua execução, o cliente terá acesso a essa informação e logo, irá tratar da melhor forma. O que não era possível na versão anterior implemetada com a API de Sockets.