Trabalho Prático - Parte 2
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 terceira 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. Nessa parte, foi desenvolvida a primeira implementação do projeto e, para isso, foi utilizado como meio de troca de informações entre processos, a API de Sockets.

2. Desenvolvimento

O sistema distribuído criado é composto por dois processos principais, o servidor e o cliente. O processo servidor é responsável por receber, processar e responder às requisições de usuários, que fazem tais requisições através de processos clientes. A seguir são apresentadas as descrições das principais funcionalidades de cada processo, e a forma em que as mensagens foram transmitidas. Além disso, são discutidas as modelagens do banco de dados, as implementações da comunicação 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, 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.

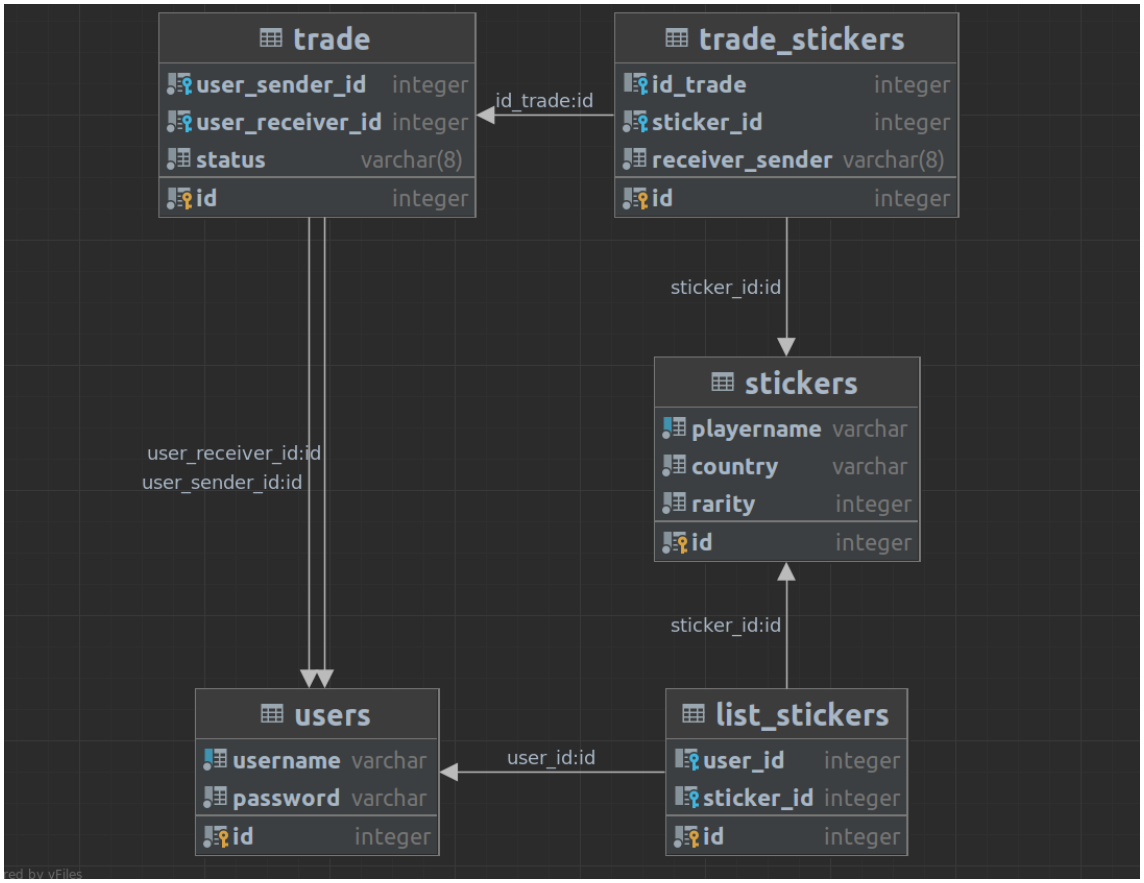

Figura 1. Digrama Relacional

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 enviada ao servidor como uma mensagem de texto que contém a descrição da operação desejada, em seguida, o processo cliente aguarda uma resposta do servidor com os resultados da operação.

2.3. Processo Servidor

O processo servidor, inicialmente cria um Socket em modo de escuta que será utilizado para trocar mensagens com algum processo cliente. Em seguida aguarda, de forma cíclica,

a conexão de um processo cliente, quando isso ocorre uma nova *thread* é criada e esta fica responsável por realizar as operações solicitadas por esse cliente.

2.4. Comunicação

A comunicação é feita através dos comandos requisição-resposta. Definimos a seguinte lista de comandos para requisição:

1. Criar usuário
2. Login
3. Listar figurinhas de um usuário
4. Solicitar troca de figurinhas
5. Listar solicitações recebidas
6. Aceitar solicitação
7. Requisitar informação de um usuário

Cada requisição é uma subclasse de comando, uma classe que possui métodos gerais: *execute* e *as_dict*, além do atributo *message_type* que é o nome da classe. O método *execute* realiza a conversão do dicionário de dados gerado pelo método *as_dict* para JSON(JavaScript Object Notation) que é uma representação externa de dados. Portanto cada requisição listada em 2.4 estão relacionadas a duas classes, uma para requisição e outra para resposta como visto em 2.

```
deneribeiro10, 4 days ago | 2 authors (deneribeiro10 and others)
class RequestTradeUserToUserCommand(Command):
    def __init__(
        self,
        user_orig: str,
        stickers_user_orig: List[int],
        user_dest: str,
        stickers_user_dest: List[int],
    ):
        self.message_type = RequestTradeUserToUserCommand.__name__
        self.user_orig = user_orig
        self.user_dest = user_dest
        self.stickers_user_orig = stickers_user_orig
        self.stickers_user_dest = stickers_user_dest


You, 3 days ago | 2 authors (You and others)
class ResponseTradeUserToUserCommand(Command):
    def __init__(self, status: bool):
        self.message_type = ResponseTradeUserToUserCommand.__name__
        self.status = status

    @classmethod
    def from_dict(cls, obj: dict):
        return ResponseTradeUserToUserCommand(status=obj["status"])
```

Figura 2. Classes de Requisição-Resposta

Portanto, essas classes tratam a representação de dados que vão serão enviadas/recebidas na troca de mensagens. Logo, temos mais um desacoplamento de funcionalidade, o servidor lerá um Comando e o cliente sempre enviará um comando. Esse

desacoplamento, facilita a escrita e a leitura de dados, pois as partes responsáveis por essas funções não precisam saber o que estão enviando, se é uma requisição para criar um usuário ou se é requisição para login, apenas sabem que é um comando.

Para o envio das mensagens, então, definimos uma classe que recebe uma *string* e envia todos os bytes de uma vez. Além disso, definimos o caractere EOF para determinar o final da mensagem. Na leitura, recebemos, no máximo, 4096 bytes por vez, portanto a definição do final da mensagem é importante para saber quando parar de receber bytes e enviar para o processador das mensagens.

Esse processador é de fácil implementação pois determinamos o tipo da mensagem como o nome da classe de requisição ou resposta, logo foi feito um esquema de verificação para saber se o tipo da mensagem era igual ao nome de alguma classe. Após essa verificação, um objeto dessa classe é criado a partir dos dados de entrada, e depois é feita a ação requisitada e posteriormente é enviada uma resposta para o cliente. A resposta tem como objetivo informar se a operação foi um sucesso ou não, ou retornar algum dado solicitado. Na figura 3 é possível ver o código dessa estratégia.

```
elif message_type == RequestAnswerTradeCommand.__name__:
    cmd = ResponseAnswerTradeCommand(False)
    try:
        self.tradestickers.answer_trade(data["trade_id"], data["accept"])
        cmd = ResponseAnswerTradeCommand(True)
    except:
        traceback.print_exception(*sys.exc_info())
```

Figura 3. Requisição de aceita/recusa de uma troca

É possível ver na linha abaixo da bloco *try* que essa operação é feita a partir de uma função, que pertence a uma classe chamada *TradeStickersService*. Desacoplamos também as operações mais custosas do envio/recebimento de mensagens, para reaproveitar a lógica para os próximos trabalhos. 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. Com o id da troca e o status de aceitar essa operação é executada com sucesso se as figurinhas estiverem com seus respectivos usuários, ou seja, suponha que $fig_1, fig_2 \in F_1$ e $fig_3 \in F_2$, sendo F_1 e F_2 os conjuntos de troca que está sendo enviado e recebido respectivamente. Se, por exemplo, a fig_1 tiver sido trocada com outro usuário essa operação será inválida e não acontecerá a troca e o servidor retorna uma resposta com *status* igual a *False*, que representa um erro, caso contrário, o servidor retorna uma resposta com *status* igual a *True*.

2.5. Threads

Além de o sistema usar uma representação externa de dados, suporta múltiplos clientes enviando mensagens para o servidor, através de *threads*. O *socket* é criado no início da execução da interface e é utilizada ao longo do uso do sistema. Em um primeiro momento, criávamos uma conexão partindo do processo cliente para o processo servidor quando era preciso requisitar algum dado, e fechávamos a conexão quando a resposta era recebida. No entanto, percebemos que não faz sentido ter *threads* quando temos poucos usuários

utilizando o sistema com esse modelo, pois a chance de um usuário realizar requisição ao mesmo tempo de outro usuário é muito baixa. Com isso, mudamos o modelo do cliente com o início da execução da interface criando uma conexão do *socket* e o encerramento da mesma quando o programa para de ser executado.

Essa mudança resultou em algumas dificuldades, estávamos criando uma mesma sessão de uso do banco de dados para todos os usuários que usassem o sistema. Isso acarreta em um mal funcionamento quando acontece uma atualização do banco de dados em um cliente, o outro usuário não percebe essa atualização. Portanto, há assincronia das sessões. Para resolver esse problema criamos uma sessão para cada usuário como mostra a figura 4, na linha 35.

```
16  class ServerSocket:
17      HOST = "127.0.0.1"
18      PORT = 5555
19
20      def __init__(
21          self,
22          session
23      ) → None:
24          self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
25          self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
26          self.sock.bind((self.HOST, self.PORT))
27
28          self.session = session
29
30      def listen(self):
31          self.sock.listen(5)
32          while True:
33              client, address = self.sock.accept()
34              print(f"Connected by {address}")
35              threading.Thread(target=self._start, args=(client,self.session(),)).start()
36
37      def _start(self, client, session):
38          s_repo = StickersRepository(session)
39          ls_repo = ListStickersRepository(session)
40          us_repo = UsersRepository(session)
41          t_repo = TradeRepository(session)
42          tr_repo = TradeStickersRepository(session)
43
44          reader_request = ReaderRequest(us_repo, s_repo, ls_repo, t_repo, tr_repo)
45
46          while True:
47              try:
48                  cmd = reader_request.read(client)
49                  Writer.write_command(client, cmd)      You, last week • bug: socket hanging up
50              except ClientDisconnectedException:
51                  print("connection closed")
52                  client.close()
53                  return
54              except Exception as e:
55                  traceback.print_exception(*sys.exc_info())
56
```

Figura 4. Servidor

A figura acima mostra como implementamos a classe servidor com *threads*.

2.6. 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 5 apresentam todas as telas descritas.

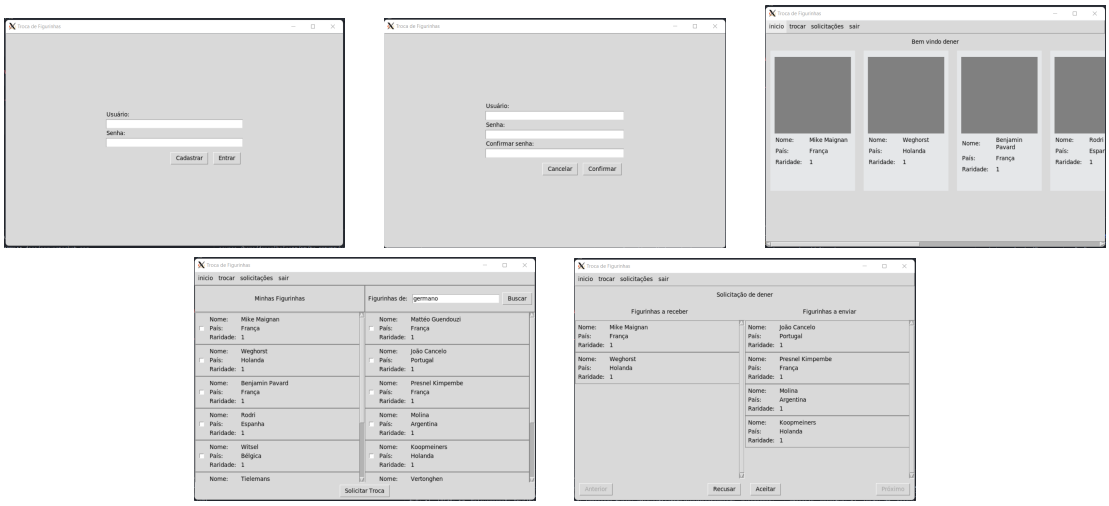

Figura 5. Telas do sisitema

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.

4. Conclusão

Em virtude do que foi apresentado, foi possível perceber que um sistema distribuído que utiliza a API de Sockets como forma de comunicação enfrenta muitas dificuldades devido à complexidade de desenvolver mensagens concisas. Além disso, foi visto na prática que uma implementação bem modularizada facilita o processo de refatoração do código, quando uma mudança na implementação se torna necessária.