



UNIVERSIDADE FEDERAL DE VIÇOSA - UFV - CAMPUS FLORESTAL

SISTEMAS DISTRIBUÍDOS E PARALELOS

Trabalho Prático 1 - Parte 3

Implementação usando API de Sockets

Emily Lopes Almeida - [Emily-Lopes](#)

Ingred Fonseca de Almeida - [ingredalmeida1](#)

Iury Martins Pereira - [iurymartins46](#)

Letícia Oliveira Silva - [leticiasilvaa](#)

Florestal - MG

2024

Sumário

1. Introdução	3
1.1. Como Executar	3
2. Desenvolvimento	4
2.1. Organização do Código	4
2.2. Decisões de Implementação	6
2.2.1. Mensagens Trocadas entre Processos	6
2.2.2. Gerenciamento de Conexões com Servidores	6
2.2.3. Controle do Estado do Jogo	7
2.3. Banco de Dados	8
2.4. Interface Gráfica	9
3. Conclusão	12
4. Referências	13

1. Introdução

O sistema distribuído desenvolvido, denominado `EmojiCards`, consiste em um jogo de cartas destinado a envolver os usuários na exploração das emoções humanas. O objetivo é proporcionar aos jogadores a oportunidade de colecionar cartas, compor seus próprios baralhos e competir em um ambiente virtual colaborativo. Como incentivo adicional, os vencedores das partidas são recompensados com uma nova carta para expandir sua coleção.

A implementação do sistema foi feita utilizando `API de Sockets`, portanto as entidades arquitetônicas são representadas por processos e a comunicação é feita utilizando o paradigma de comunicação direta entre processos. Além disso, toda comunicação entre processo foi feita por fluxo `TCP`.

1.1. Como Executar

Para facilitar a execução do sistema distribuído foi utilizado o Docker. Portanto, antes de executar é preciso ter o Docker configurado na máquina.

No Sistema Operacional Linux, primeiramente, é preciso executar o comando abaixo antes de executar os containers para liberar o acesso do display ao Docker:

```
$ xhost +local:*
```

Se for a primeira execução do sistema, é preciso usar o comando abaixo para preparar as imagens remotas:

```
$ docker compose build
```

Em seguida, basta utilizar o comando abaixo para iniciar os containers descritos no ``docker-compose.yml``:

```
$ docker compose up
```

Com isso, o terminal Docker será iniciado e para encerrá-lo basta fazer `Ctrl + C`. Já para destruir os containers é preciso utilizar o comando:

```
$ docker compose down
```

O arquivo ``docker-compose.yml`` tem um contêiner para o servidor de banco de dados, um contêiner para o servidor de aplicação e três contêineres para três clientes, já que para jogar uma partida é preciso ter no mínimo três jogadores. Portanto, ao executar os comandos descritos, os dois servidores estarão disponíveis e três jogadores serão iniciados e por isso, três janelas com a interface gráfica serão abertas na página de login. Também para facilitar, no banco de dados já existem três usuários cadastrados:

username	senha
thais	12345678
henrique	12345678
aluno	12345678

2. Desenvolvimento

2.1. Organização do Código

Com o objetivo de desenvolver um sistema robusto e modularizado, foi definido um modelo de arquitetura baseado em três camadas físicas e quatro camadas lógicas. As camadas físicas são distribuídas em apresentação, aplicação e dados, garantindo uma separação clara entre a interface do usuário, a lógica de negócios e o armazenamento de informações. As camadas lógicas, por sua vez, incluem: apresentação, negócio, comunicação e plataforma, sendo que, vale destacar que a camada lógica de apresentação, que corresponde à interface gráfica do usuário, está presente apenas no cliente, permitindo a interação direta do usuário com o sistema.

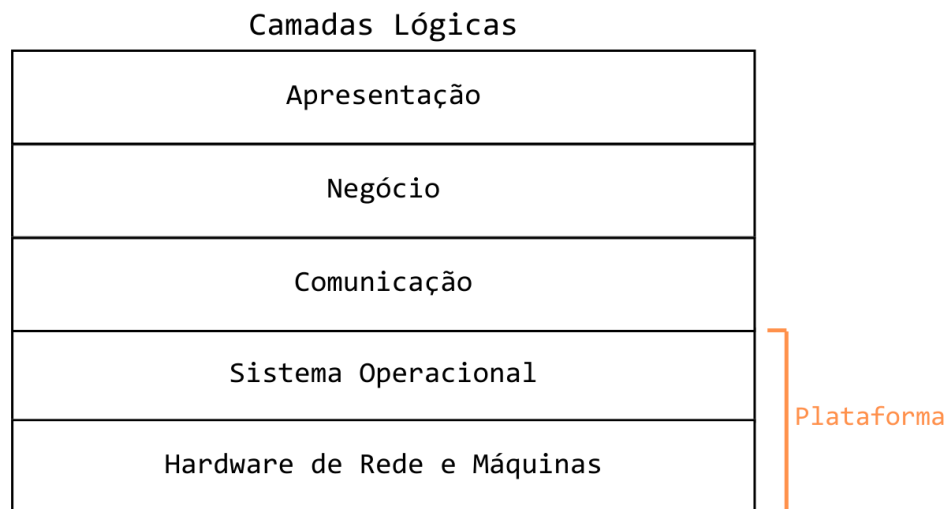


Figura 1 - Diagrama Representativo das Camadas Lógicas

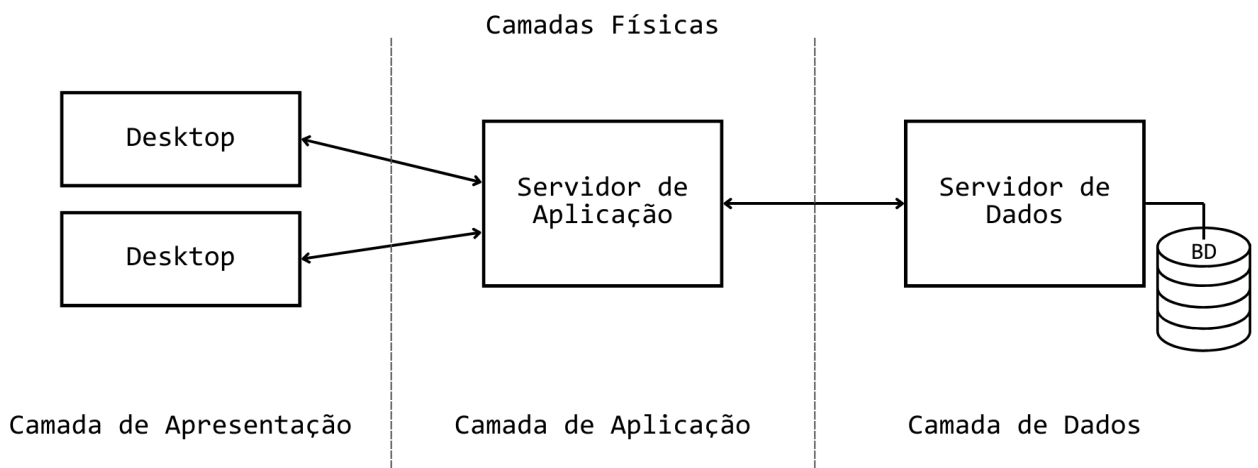


Figura 2 - Diagrama Representativo das Camadas Físicas

Nesse sentido, como mostra a Figura 3, o código foi organizado de forma a ter uma pasta representando o servidor de aplicação (`/server-app`), uma pasta representando o servidor de banco de dados (`/server-bd`), e uma pasta representando um cliente (`/cliente`). Com relação às camadas lógicas, no servidor de aplicação a camada de comunicação é representada pelo arquivo `server.py` e o arquivo `funcionalidades.py` representa a camada de negócio.

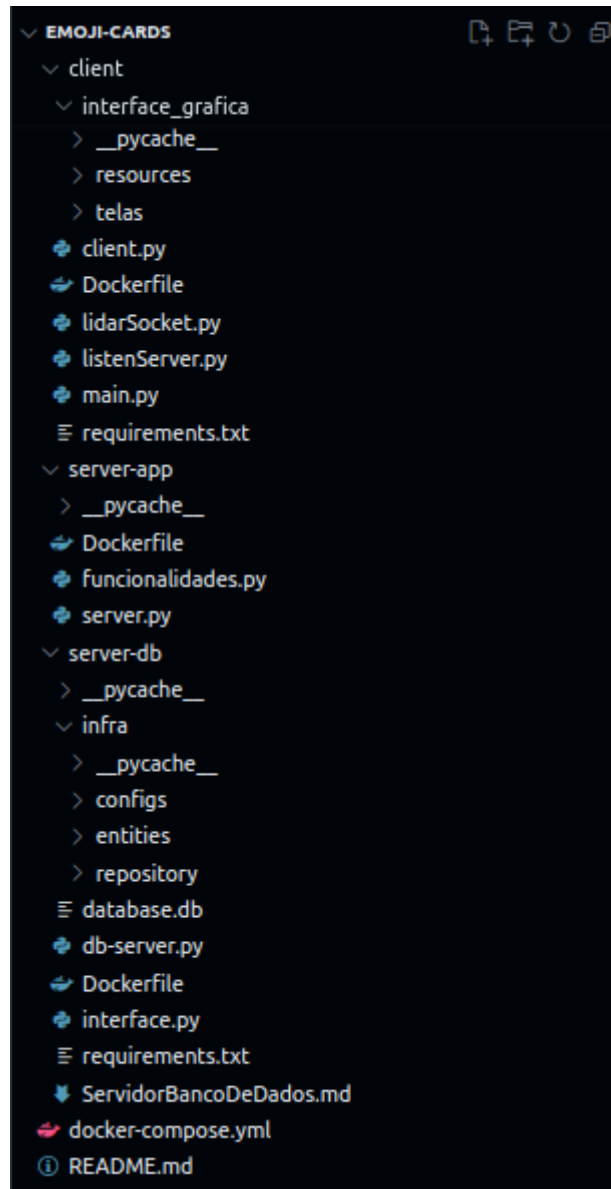


Figura 3 - Arquivos e Pastas do Projeto

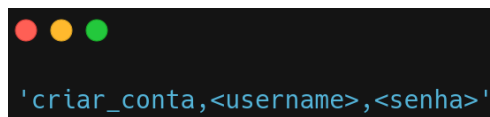
No servidor de banco de dados, a camada de comunicação é representada pelo arquivo `db-server.py` e a camada de negócio é representada pelo arquivo `interface.py` e pela pasta `infra`, de forma que dentro dessa pasta temos as configurações (pasta `/configs`), tabelas, (pasta `/entities`) e interfaces diretas para acesso ao conteúdo das tabelas (pasta `/repository`). O arquivo `interface.py` é uma interface intermediária entre a camada de comunicação e a interface de comunicação direta com os dados armazenados.

Já no cliente, a camada de comunicação é representada pelos arquivos `lidarSocket.py` e `listenServer.py`, o negócio é representado pelo arquivo `client.py` e a apresentação é representada pela pasta `/interface_gráfica`.

2.2. Decisões de Implementação

2.2.1. Mensagens Trocadas entre Processos

Antes de iniciar a implementação do sistema, o fluxo de comunicação entre as camadas físicas do sistema foi elaborado para facilitar o entendimento do sistema como um todo. Durante esse processo foi discutida a importância de definir o formato das possíveis requisições de forma a facilitar a troca e a manipulação das mensagens, e também a integração do sistema. A primeira decisão tomada foi definir que a primeira parte da requisição sempre seria o identificador da mesma. Essa escolha permite que cada mensagem seja imediatamente reconhecida e processada conforme o contexto, chamando as funções necessárias para processar a requisição.



```
'criar_conta,<username>,<senha>'
```

Figura 4 - Exemplo da requisição recebida pelo servidor para criar uma nova conta

Além disso, todas as mensagens são estruturadas como `strings` visto que o Python oferece diversas possibilidades para a manipulação eficiente delas, como o método `split()`, que divide a string em partes menores, convertendo-a em um vetor de elementos e o método `eval()` que permite a conversão da string em um dicionário, que é uma alternativa oferecida pelo Python que simula um arquivo `Json` e que facilita o acesso e a organização dos dados,.

2.2.2. Gerenciamento de Conexões com Servidores

O sistema `EmojiCards` foi projetado para ser multiusuário, sendo assim, tanto o servidor de aplicação quanto o servidor banco de dados precisam ser capazes de lidar com requisições vindas de diferentes máquinas e precisam estar sempre disponíveis para receber novas requisições. Nesse contexto, para manter o sistema responsivo, foram utilizadas `Threads`, de modo que o servidor cria uma nova `thread` para cada requisição recebida. Dessa forma, a `thread` principal, associada à porta conhecida para comunicação entre processos e o servidor, permanece sempre disponível para receber novas requisições, evitando bloqueios e garantindo uma performance consistente mesmo sob demanda de múltiplos usuários.



```
def start_server():
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind(("0.0.0.0", 5000))
    server.listen()
    print("servidor ouvindo na porta 5000...")

    while True:
        client_socket, addr = server.accept()
        print(f"conexão aceita de {addr}")

        client_handler = threading.Thread(target=handle_client, args=(client_socket,))
        client_handler.start()

if __name__ == "__main__":
    start_server()
```

Figura 5 - Inicialização do Servidor e Criação de Threads para Gerenciamento de Requisições

2.2.3. Controle do Estado do Jogo

Assim que um usuário faz login no sistema, uma conexão é estabelecida, permitindo que o servidor identifique a porta e o endereço IP pelos quais pode se comunicar com o usuário. Isso é importante para que o servidor de aplicação possa enviar mensagens para os usuários, como convites para partidas ou atualizações sobre o turno de uma partida em que o usuário está envolvido. Esse canal de comunicação é utilizado exclusivamente pelo servidor de aplicação.

Quando o cliente precisa enviar mensagens ao servidor, novas conexões são estabelecidas através de *threads* dedicadas para lidar com cada requisição específica. O servidor pode responder por meio dessa nova conexão temporária ou pela conexão já existente, que é utilizada para mensagens relacionadas ao controle da partida.

Para gerenciar os usuários online, o servidor utiliza um dicionário que associa cada nome de usuário ao *socket* correspondente. Quando o usuário faz o logout do sistema, o servidor remove automaticamente o usuário do controle de usuários ativos.

```
def login(self, username, senha):
    try:
        client = self.criar_conexao()
        request = f"login,{username},{senha}"
        self.enviar_dados(client, request)
        response = self.receber_dados(client)
        port, response = response.split(",")
        if response == "Login feito com sucesso!":
            self.username = username
            client_ip, client_port = client.getsockname()
            client_port = port
            client.close()
            listen_thread = threading.Thread(target=self.handle_server, args=(client_ip, client_port))
            listen_thread.daemon = True
            listen_thread.start()
            return True, response
        return False, response
    except Exception as e:
        self.fechar_conexao(client)
        print(f"Erro ao realizar login:\n{e}")
        return False, None
```

Figura 6 - Criação da Thread para Escutar Mensagens do Servidor direcionadas ao Cliente

```
def handle_server(self, client_ip, client_port):
    try:
        server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        server.bind(('0.0.0.0', int(client_port)))
        server.listen(1)

        print(f"Cliente escutando em {client_ip}:{client_port}\n")

        while True:
            try:
                server_socket, addr = server.accept()
                print(f"Conexão recebida de {addr}")
                request = self.receber_dados(server_socket)
                if request:
                    thread = threading.Thread(target=self.__manipular_mensagem, args=(request,))
                    thread.start()

            except Exception as e:
                print(f"Erro ao ouvir mensagem de convite: \n{e}")
    except Exception as e:
        print(f"Erro ao tentar escutar servidor {client_ip}:{client_port} \n{e}")
```

Figura 7 - Canal de Comunicação com Usuário Disponível ao Servidor de Aplicação

Com relação às informações de uma partida, por serem informações que só têm relevância enquanto a partida está em andamento, elas são armazenadas pelo servidor de aplicação, em um dicionário, para permitir uma gestão mais ágil e direta. Essa abordagem evita a necessidade do servidor de aplicação aguardar respostas do banco de dados a cada interação, o que poderia introduzir latência e comprometer a experiência do usuário.

Ao final da partida, caso haja um vencedor, é preciso atualizar o banco de dados incluindo uma nova carta na coleção do usuário vencedor, realizando nesse momento a requisição específica ao servidor de banco de dados para efetivar essa atualização. Em seguida, a partida pode ser simplesmente removida do dicionário no servidor de aplicação, sem a necessidade de operações adicionais no banco de dados.

2.3. Banco de Dados

O banco de dados do sistema distribuído foi implementado utilizando SQLite com arquivo. Com o objetivo de abstrair a camada de banco de dados, utilizou-se a biblioteca SQLAlchemy, especialmente a funcionalidade ORM que permite interagir diretamente com objetos Python para realizar operações de banco de dados, abstraindo a maior parte das interações SQL. Isso facilita a manutenção e a evolução do código, permitindo modificações no esquema de dados sem a necessidade de reescrever consultas SQL manualmente. Além disso, o uso de ORM permite uma independência de banco de dados pois a abstração proporcionada pelo SQLAlchemy permite que o sistema seja facilmente migrado para outros SGDBs, caso seja necessário.

O banco de dados é estruturado com duas tabelas principais: `Usuario` e `Carta`. As tabelas abaixo ilustram os atributos de cada tabela juntamente de um exemplo de como os dados são registrados.

Tabela Carta		
atributos		exemplo
emocao	[String, Chave Primária]	autoestima
atributos	[String]	{ 'tempo': 'anos', 'impacto_social': 'positivo', 'efeito_cognitivo': 'certeza', 'qtd_emocoes_opostas': 1, 'qtd emoções relacionadas': 1, 'intensidade': 5}

Tabela Usuario		
atributos		exemplo
username	[String, Chave Primária]	thais
senha	[String]	12345678
status	[String]	jogando
colecacao_cartas	[String]	autoestima,..., raiva
qtd_baralhos	[Integer]	2
baralhos	[String]	paz,paz,,..., euforia-paz,..., raiva

A forma de armazenamento dos dados foi escolhida para facilitar a comunicação eficiente dentro do sistema. Na tabela `carta`, a coluna `emocao` é usada como chave primária, o que permite que a interface encontre a imagem da carta correspondente através do nome da emoção, uma vez que os arquivos das imagens foram nomeados seguindo o formato `<emocao>.png`, o que facilita a recuperação da imagem associada.

Os atributos das cartas são armazenados em uma única coluna como uma string `Json` que simula um dicionário. Dessa forma, servidor de aplicação pode converter a string utilizando o método `eval()` em um dicionário para acessar o valor do atributo usando a chave correspondente, otimizando a manipulação dos dados.

Para a tabela usuário, o status de um usuário pode ser "offline", "online" ou "jogando". Essa informação é importante para o servidor de aplicação conferir se pode enviar um convite para um jogador, o que só pode ser feito se o usuário estiver "online". A coleção de cartas é armazenada como uma string com os nomes das emoções separados por vírgulas. Essa string pode ser facilmente convertida em uma lista usando o método `split()`, permitindo à interface localizar a carta pelo nome do arquivo e ao servidor de aplicação buscar os atributos da carta. Da mesma forma, o baralho é armazenado como uma string com emoções separadas por vírgulas, e os diferentes baralhos são separados por um delimitador "-", de forma que os baralhos também possam ser separados e processados utilizando o método `split()`.

2.4. Interface Gráfica

Para desenvolver a interface gráfica do sistema, utilizou-se a biblioteca Arcade que é uma biblioteca utilizada para a criação de jogos e aplicações gráficas 2D em Python. Foram desenvolvidas telas para Login/Cadastro, Visualizar Perfil, Responder Convite para Partida, Montar Baralho, Criar Partida, Aguardar Jogadores, Escolher Baralho, Escolher Carta do Turno, Informar Vencedor do Turno e Informar Vencedor da Partida.

A interface é inicializada a partir do arquivo `main.py` da pasta `/client` que é referenciado no `Dockerfile`. Ele é responsável por criar uma janela e inicializar a sequência de telas, que como apresentado na Figura 8, inicia na tela Login, onde o jogador insere seus dados previamente cadastrados ou, caso o jogador ainda não tenha uma conta, cria uma nova conta. Feito isso, o jogador é direcionado para a tela Criar Partida.

Antes de criar uma partida é importante que o usuário crie um baralho a partir da sua coleção de cartas acessando a tela do seu Perfil, para que ele possa escolher com qual conjunto de cartas irá jogar determinada partida, caso contrário o sistema fornecerá um baralho aleatório gerado a partir da coleção do jogador.

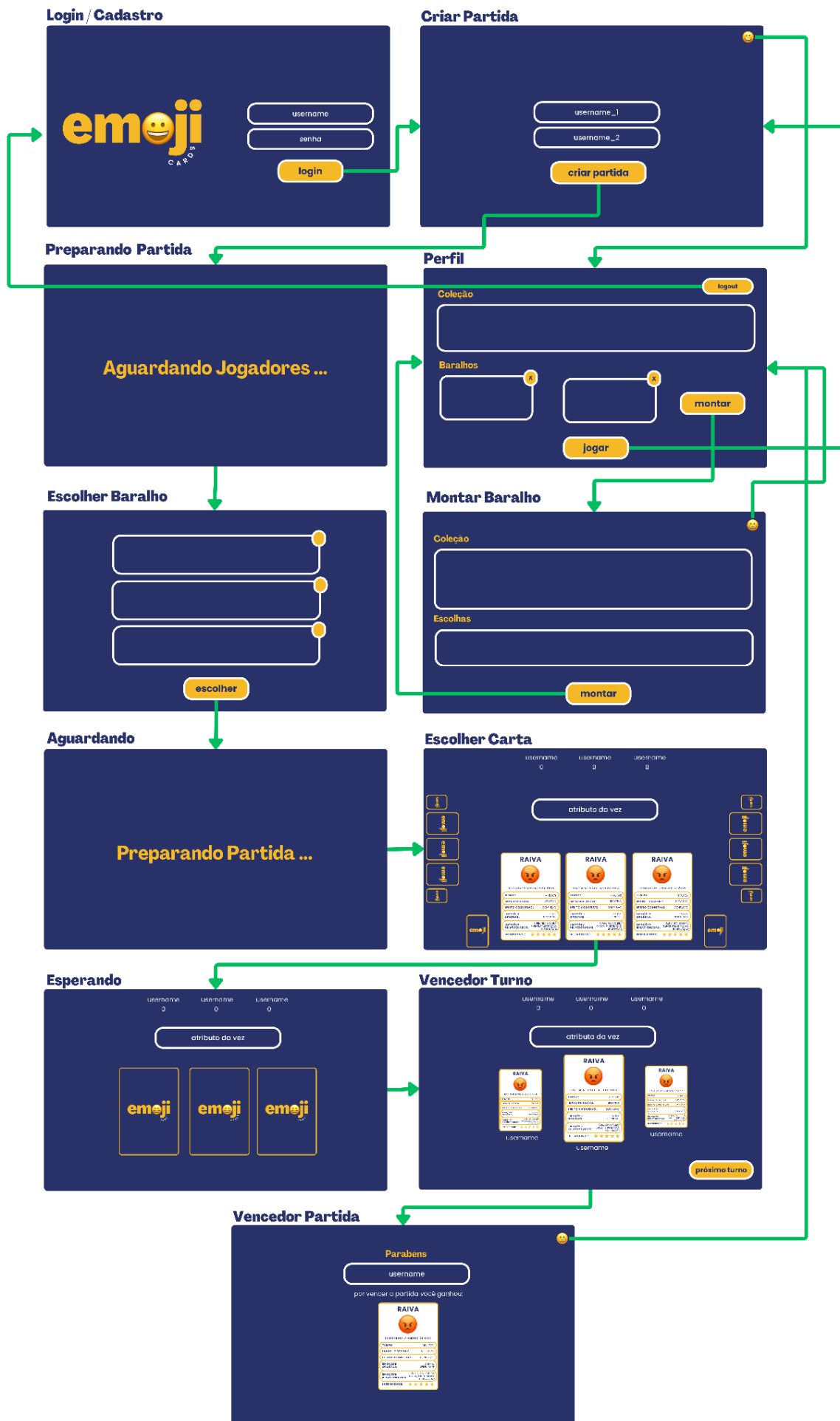


Figura 8 - Fluxo Geral das Telas

Além de criar uma partida, o jogador pode receber convites para partidas criadas por outros jogadores, para isso foi criado uma tela simulando um pop-up (figura 9) para apresentar o convite ao jogador enquanto ele não está em outra partida e se ele estiver na tela criar partida quando o convite é recebido.



Figura 9 - Pop-up do convite para partida

Se o jogador criou uma partida ou aceitou o convite de outro jogador ele será direcionado para a tela de escolher seu baralho e posteriormente a partida em si, formada de sete turnos em que o usuário recebe três cartas do seu baralho e escolhe uma delas para concorrer com as cartas escolhidas pelos outros jogadores, de acordo com o atributo gerado aleatoriamente para o turno. Assim que todos os jogadores escolhem suas cartas, o sistema apresenta o vencedor do turno e inicia um novo turno com outro atributo e outro conjunto de três cartas para cada jogador. Esse processo se repete sete vezes até que o último turno ocorra, quando será possível que o sistema determine o vencedor da partida e qual carta será adicionada a sua coleção.

Para garantir que a interface respondesse corretamente às mensagens recebidas do servidor, exploramos o método `on_update()`, já definido por padrão para telas na biblioteca. Esse método verifica se houve alguma mudança no estado dos atributos da tela e atualiza a interface conforme as novas especificações. Cada tela foi estruturada para ter como atributo uma instância da classe `Client`, que representa a camada de negócio do cliente, e essa instância, por sua vez, possui um atributo destinado a armazenar mensagens recebidas do servidor pelo canal de comunicação disponível. Assim, ao receber uma nova mensagem, a interface detecta a alteração e realiza as atualizações necessárias de forma automática.

3. Conclusão

Durante o desenvolvimento do sistema `EmojiCards` com programação via `sockets`, foram enfrentadas algumas dificuldades. Uma das principais foi relacionada ao estabelecimento do canal de comunicação entre cliente e servidor no momento em que um cliente envia uma requisição de login, pois em alguns momentos, o cliente deixava de escutar na porta designada, resultando na perda da conexão. Essa situação tornou-se complexa, pois estava impedindo a simulação de clientes em diferentes processos, já que um cliente poderia usar a porta de outro, uma vez que tudo estava sendo executado como um único processo.

Para resolver esse problema, foi implementado um mecanismo de `thread` no cliente, permitindo que ele permanecesse escutando na mesma porta em que a requisição de login foi enviada. Além disso, o servidor passou a manter um registro ativo dos clientes conectados e foram criados diferentes arquivos para simular processos distintos. Embora compartilhassem o mesmo endereço IP, esses processos eram alocados em diferentes portas. Essa abordagem permitiu que o sistema recuperasse conexões perdidas e garantisse que as requisições fossem corretamente direcionadas.

Outra dificuldade significativa foi relacionada ao desempacotamento das requisições quando estas continham muitos atributos. O processo de extração e interpretação de dados complexos resultou em erros, dificultando a implementação de funcionalidades. Para resolver esse problema, optamos por padronizar as mensagens enviadas entre o cliente e o servidor, adotando um formato estruturado e conhecido entre os dois. Essa decisão não apenas simplificou o desempacotamento dos dados, mas também tornou o código mais legível e fácil de manter, facilitando a inclusão de novos atributos quando necessário.

O desenvolvimento da interface gráfica também foi desafiador. Além da dificuldade em aprender a usar uma nova biblioteca e entender suas funcionalidades, foi necessário elaborar estratégias para garantir que a interface respondesse adequadamente às mensagens recebidas do servidor. Esse processo demandou um tempo considerável, especialmente devido à complexidade do jogo, que exigia a criação e gerenciamento de várias telas, além da dependência da interação entre três usuários.

Por fim, apesar dos desafios, foi possível realizar a proposta do trabalho e alcançar o objetivo principal de implementar um sistema distribuído utilizando `sockets`. Cada obstáculo superado contribuiu para o aprimoramento das habilidades dos integrantes do grupo e para o desenvolvimento de um sistema mais robusto e funcional.

4. Referências

- [1] Um guia completo para programação de sockets em Python. Disponível em: <https://www.datacamp.com/tutorial/a-complete-guide-to-socket-programming-in-python>
- [2] SQLAlchemy. Disponível em: <https://docs.sqlalchemy.org/en/20/>
- [3] Guia SQLAlchemy. Disponível em: <https://www.youtube.com/watch?v=to39SFUxOpg&list=PLAqbpJQADBGKbwhOvd9DVWy-xhA1KEGm1&pp=iAQB>
- [4] Arcade. Disponível em: <https://api.arcade.academy/en/latest/>