UNIVERSIDADE FEDERAL DE VIÇOSA - UFV - CAMPUS FLORESTAL

SISTEMAS DISTRIBUÍDOS E PARALELOS

Trabalho Prático 3

Miguel Antônio Ribeiro e Silva - 4680

João Victor Graciano Belfort de Andrade - 4694

Paula Teresa Mota Gibrim - 4234

2024

1. Introdução

O trabalho iniciou-se com uma revisão e reformulação das ideias das partes 1 e 2 do projeto. Identificamos algumas lógicas que precisavam ser ajustadas e adaptamos nossa proposta para melhorar a funcionalidade e a eficiência do sistema.

Focamos em quatro pontos principais durante o desenvolvimento:

1. **Banco de Dados:** Criamos o banco de dados para garantir que a modelagem e o gerenciamento das informações fossem adequados às necessidades do projeto.
2. **Lógica do Jogo:** Para que todas as regras e funcionalidades fossem implementadas corretamente e que a experiência do usuário fosse a melhor possível.
3. **Arquitetura do projeto**: A parte que mais mandou dedicação, criamos uma arquitetura única.
4. **Interface do Usuário:** Desenvolvemos a interface do usuário para garantir que fosse intuitiva e funcional.

Para rodar o nosso projeto, utilize o comando **make run** na pasta **src** do projeto. Caso por algum motivo falte a instalação de alguma biblioteca, basta instalar manualmente como indicará o erro. Se o comando **make run** não funcionar, basta rodar os arquivos separados: **main.py, main2.py**, **main3.py** e **run_server.py** no seu próprio ambiente.

A estrutura do projeto segue o padrão **Modelo-Visão-Controle (MVC)**, que organiza o código em três camadas distintas:

- **Modelo:** Responsável pela gestão dos dados e lógica de negócio.
- **Visão:** Cuida da interface do usuário e da apresentação dos dados.
- **Controle:** Faz a mediação entre o modelo e a visão, processando as entradas do usuário e atualizando o modelo e a visão conforme necessário

O projeto está organizado em várias pastas para facilitar a manutenção e o desenvolvimento. As principais pastas incluem:

- **assets**: contém os arquivos de fontes e imagens utilizadas na interface do jogo.
- **controller**: abriga o controlador do cliente com funções como **client_connection**, que é crucial para a interação com o servidor.
- **model**: contém as classes e a gestão de dados, incluindo modelos como **User**, **Deck** e operações relacionadas ao banco de dados.
- **server**: guarda o controlador do servidor, com funções como **server_connection** para gerenciar as conexões com os clientes.
- **view**: apresenta uma interface básica para o usuário, com menus e opções de interação.

Na pasta **src**, temos três arquivos principais (**main1.py**, **main2.py**, **main3.py**), cada um para um cliente específico. O projeto também inclui o banco de dados **client.db**, o arquivo de dependências **requirements.txt**, e o script **run_server.py** para executar o servidor. Por fim, o **Makefile** é utilizado para facilitar a compilação e execução do projeto

2 Desenvolvimento

2.1 Banco de Dados

Como solicitado, nosso sistema utiliza o **SQLite** para gerenciamento de dados, armazenando as informações no arquivo **client.db**. O banco de dados é estruturado da seguinte forma:

Estrutura das Tabelas

1. **Tabela cards**
    - Armazena informações sobre as cartas do jogo de trunfo.
    - **Colunas:**
        - **id**: Identificador único da carta.
        - **name**: Nome da carta.
        - **intelligence**: Valor do atributo inteligência.
        - **charisma**: Valor do atributo carisma.
        - **sport**: Valor do atributo esporte.
        - **humor**: Valor do atributo humor.
        - **creativity**: Valor do atributo criatividade.
        - **appearance**: Valor do atributo aparência.
2. **Tabela client**
    - Contém os dados dos jogadores.
    - **Colunas:**
        - **id**: Identificador único do jogador.
        - **name**: Nome do jogador.
3. **Tabela client_id_card_id**
    - Relaciona um jogador a suas cartas, formando um "baralho" completo para cada jogador.
    - **Colunas:**
        - **client_id**: Identificador do jogador.
        - **card_id**: Identificador da carta.
4. **Tabela deck**
    - Define o deck específico que um jogador monta para a partida.
    - **Colunas:**
        - **id**: Identificador único do deck.
        - **card_id**: Identificador da carta.
        - **client_id**: Identificador do jogador.

Distinção entre Baralho e Deck

- **Baralho:** Refere-se a todas as cartas que um jogador possui no sistema.
- **Deck:** É um subconjunto do baralho, composto por até 8 cartas selecionadas pelo jogador para uso em uma partida específica.

Funcionamento do Sistema com o Banco de Dados

1. **Login e Criação de Usuário:**

    - Ao iniciar o sistema, o usuário deve selecionar um nome de usuário para logar dos predefinidos ou criar um usuário. Não há um sistema de login formal; basta clicar no nome do usuário para carregá-lo.
    - Na criação de um novo usuário, seis cartas padrão são adicionadas ao seu baralho, representando os cinco professores da UFV: **Thais Regina**, **Gláucia Braga**, **José Nacif**, **Fabrício Silva** e **Daniel Mendes** e do monitor da disciplina **Henrique Santana**, e mais quatro do baralho padrão do jogo. A carta **Thais Regina** é o SUPERTRUNFO, a carta mais forte do jogo possuindo todos os atributos 99, a do **Henrique** por sua vez é a SEMI-SUPERTRUNFO com todos os atributos 88. Além dessas cartas o usuário poderá criar sua própria carta. Por fim, seis dessas são adicionadas ao deck.
2. **Criação de cartas**: Após criar um usuário, ele deve criar sua própria carta com as seguintes regras:
    - Cada atributo pode ter um valor máximo de 10.
    - A soma de todos os atributos não pode ultrapassar 30.

    Por enquanto, o usuário pode criar várias cartas, além da sua, caso deseje.

3. **Gerenciamento de Cartas:**
    - **Visualização de Cartas:** O usuário pode visualizar todas as suas cartas clicando no botão "Print All Cards".
    - **Edição do Deck:** O botão "Edit Deck" é a principal ferramenta para a personalização do deck. O usuário pode:
        - Selecionar cartas para compor o deck que será utilizado na partida.
        - Excluir cartas do deck.
        - Visualizar o deck atual.
4. **Logout:**
    - Um botão de logout está disponível para que o usuário possa sair do sistema.

2.2 Regras e Mecânica do Jogo

Nosso jogo envolve três jogadores, cada um iniciando com um deck de cartas. O jogo se desenvolve da seguinte forma:

Início do Jogo

1. **Conexão e Preparação:**
    - Após a conexão de todos os três jogadores ao servidor, o jogo inicia.
    - Cada jogador recebe uma mão inicial composta por 5 cartas aleatórias retiradas de seu próprio deck.

Estrutura do Jogo

2. **Divisão dos Turnos:**
    - O jogo é estruturado em 5 turnos.
    - A vitória da partida é determinada pelo número de turnos ganhos, sendo o jogador com mais vitórias ao final dos 5 turnos o vencedor da partida.
3. **Desenvolvimento dos Turnos:**

    - O servidor sorteia um atributo específico para a rodada. Esse atributo é o critério pelo qual as cartas serão avaliadas.
    - Em cada turno, cada jogador escolhe uma carta de sua mão e a joga na mesa.
    - As cartas jogadas são então processadas com base no atributo sorteado. O jogador cuja carta possui o **maior** valor no atributo **selecionado** vence o turno.
    - No **caso** de **empate** (ou seja, se duas ou mais cartas tiverem o mesmo valor no atributo sorteado), todos os jogadores envolvidos no empate recebem um ponto para aquele turno.
4. **Atualização das Mãos e Pilha de Cartas:**
    - Após o primeiro turno, cada jogador terá **4 cartas** restantes em sua mão.
    - As cartas utilizadas durante o turno são movidas para a **pilha** de **cartas**, que é uma área de descarte comum a todos os jogadores.
    - A mecânica do jogo continua com os jogadores escolhendo novas cartas para jogar, enquanto as cartas já utilizadas são adicionadas à pilha.
5. **Conclusão do Jogo:**
    - Após os 5 turnos, o jogador com o maior número de turnos ganhos é declarado vencedor.
    - O vencedor da partida tem o direito de escolher uma **carta** da **pilha** de **cartas**, que inclui as cartas utilizadas por todos os jogadores durante o jogo.
    - Essa carta escolhida é adicionada à coleção do vencedor, proporcionando-lhe a oportunidade de obter cartas de outros jogadores e fortalecer seu próprio deck.

Detalhes Adicionais

- **Atributos da Rodada:** O atributo sorteado para cada rodada pode ser qualquer característica das cartas, como inteligência,esporte, ou qualquer outro critério específico definido no jogo.
- **Empate:** Em casos de empate, todos os jogadores empatados são premiados com um ponto para garantir uma competição justa e equilibrada. Em caso de empate na partida, ninguém ganha nada.
- **Pilha de Cartas:** A pilha de cartas serve como um pool comum de cartas usadas, e sua gestão é importante para garantir que o vencedor tenha acesso a uma variedade de cartas ao final da partida.

2.3 Arquitetura do Sistema Distribuído

A arquitetura desenvolvida pelo grupo é uma forma híbrida de arquitetura para sistemas distribuídos. Inicialmente, adota o modelo **cliente-servidor** tradicional, no qual o cliente envia dados ao servidor, que atua como intermediário. No entanto, após o servidor encaminhar os dados, os clientes assumem o papel de processadores, funcionando como servidores para processar as informações e executar as ações necessárias. Esse tipo de arquitetura combina aspectos de **cliente-servidor** com **peer-to-peer** (**P2P**), onde os clientes têm a capacidade de interagir ‘diretamente’ e processar dados. Portanto, essa arquitetura pode ser descrita como uma arquitetura cliente-servidor com processamento distribuído ou **cliente-servidor híbrido**.

A **comunicação** em nossa arquitetura é **bidirecional**, permitindo que tanto o cliente quanto o servidor iniciem a comunicação e respondam às mensagens recebidas. Ambos têm a capacidade de atuar como remetente e receptor, além de operar em uma **rede LAN**. Não garantimos o funcionamento em LAN em uma rede fechada, como a da UFV.

Em resumo, nossa arquitetura é uma QUASE-PEER-TO-PEER, ou seja os clientes interagem entre si e eles mesmos processam dados um dos outro. O server atua somente como uma ponte.

Funcionamento Geral

1. **Inicialização do Servidor**
    - O servidor é iniciado e aguarda conexões em várias portas.
2. **Conexão do Cliente**
    - Para ingressar em um jogo, o cliente deve clicar no botão "**Find Server**". Na tela aparecerá uma lista de servidores disponíveis para seleção. Após escolher um servidor, o cliente se conecta a ele e envia seus dados. O cliente também pode visualizar quem está conectado no servidor escolhido, bastando para isso estar conectado a pelo menos três pessoas. Isso funciona da seguinte maneira:
        - O cliente envia uma mensagem de **HANDSHAKE** para todos os IPs na rede local (**Função get_local_ips()**) e aguarda uma resposta.
        - O servidor responde com a porta utilizada para a conexão inicial. Em seguida, o servidor muda a porta e envia uma mensagem de **CONNECT** ao cliente. A partir desse momento, o cliente está apto a escutar o servidor e enviar dados. O cliente também inicia uma nova *thread* para o processamento reverso, passando a atuar simultaneamente como cliente e servidor.
3. **Função get_local_ips()**
    - Consiste em duas funções que trabalham em conjunto para obter e listar todos os endereços IP utilizáveis dentro de uma rede local. A função **get_ip_and_netmask()** utiliza o módulo **psutil** para acessar informações das interfaces de rede disponíveis no sistema. Ela itera sobre as interfaces e seus endereços associados, filtrando aqueles que são do tipo IPv4 (usando **socket.AF_INET**) e excluindo endereços de loopback (**127.x.x.x**). Quando um endereço válido é encontrado, a função retorna esse endereço IP junto com a máscara de sub-rede correspondente. Caso contrário, retornará **None** para ambos. A segunda função, **get_local_ips()**, utiliza o IP e a máscara de sub-rede obtidos para criar uma representação da rede local através do módulo **ipaddress**. Com a rede criada, a função gera uma lista de todos os endereços IP utilizáveis na faixa, excluindo os endereços reservados para rede e broadcast.
4. **Processo de Conexão**
    - Quando um servidor é encontrado em um IP específico e está em execução, ele é exibido na tela para que o cliente possa conectar-se a ele.
5. **Estrutura de Comunicação**
    - Optamos por uma estrutura de comunicação onde o cliente utiliza uma porta para enviar dados ao servidor e outra porta para receber dados. O servidor, por sua vez, também possui uma porta para receber dados e outra para enviar dados. Ao todo, são usadas quatro portas: duas para comunicação e duas para recebimento, resultando em quatro threads.
6. **Início do Jogo**

- Quando o servidor recebe os dados de conexão dos três clientes, ele começa o jogo da seguinte forma:
  - Cada jogador envia ao servidor uma mensagem compactada denominada **PLAYER_DATA**, contendo informações como nome, ID, deck, IP e porta de escuta. Com os dados de todos os três jogadores, o servidor compacta essas informações em uma estrutura **GameData** em uma nova mensagem, denominada **START_GAME**, que contém os IDs, nomes e 5 cartas escolhidas aleatoriamente dos decks de todos os jogadores. Até este ponto, o servidor atuou como "servidor" e passará a ser intermediário.
  - O servidor também envia uma mensagem **NEW PLAYER** para informar aos outros clientes sobre a entrada de um novo jogador.
  - Os clientes agora assumem o papel de "servidores" e processam tudo em suas próprias máquinas.

7. **Execução do Jogo**
   - Após receber a mensagem **START_GAME** com a **GameData**, os clientes iniciam o jogo. O servidor também envia a mensagem **ATRIBUTO** com o atributo da rodada. Cada cliente exibe ao usuário uma mão de cartas aleatória retirada do seu deck e permite que o usuário escolha uma carta para jogar. O cliente envia uma mensagem compactada **PLAY** ao servidor com a carta escolhida.
   - O servidor, por sua vez, não processa as jogadas. Em vez disso, compacta as jogadas em uma mensagem **PLAY** com as cartas escolhidas por cada jogador e seus respectivos IDs. Os clientes processam essas jogadas em suas máquinas, determinando o vencedor de cada turno. Esse processo se repete por cinco turnos.
8. **Conclusão do Jogo**
   - Ao final do jogo, os clientes enviam ao servidor uma mensagem **WINNER** perguntando quem venceu a partida. O servidor responde a todos os clientes com o vencedor da partida. O vencedor, por sua vez, tem a opção de escolher uma carta da **pilha** para compor sua coleção.
   - O cliente vencedor envia ao servidor uma mensagem de **ENCERRAR**, que é encaminhada a todos os clientes para encerrar o jogo.
9. **Mensagens Especiais**
   - **HANDSHAKE**: Mensagem inicial enviada pelo cliente para descobrir servidores disponíveis na rede.
   - **CONNECT**: Mensagem enviada pelo cliente após receber a porta de descoberta, para estabelecer a conexão com o servidor.
   - **PLAYER DATA**: Mensagem contendo informações do jogador (nome, ID, deck, IP e porta de escuta).
   - **NEW PLAYER**: Mensagem enviada pelo servidor para notificar os clientes sobre a entrada de um novo jogador.
   - **DISCONNECT**: Mensagem enviada pelo cliente para informar ao servidor que ele está saindo do jogo, encerrando a partida.
   - **TYPO_ERROR**: Mensagem indicando um erro de digitação ou outro erro na comunicação.
   - **START_GAME**: Mensagem do servidor para iniciar o jogo, incluindo os dados de todos os jogadores.
   - **PLAY**: Mensagem contendo a jogada de um jogador, que é compactada e enviada a todos os clientes.

- **ATRIBUTO**: Mensagem contendo um dos 6 atributos, enviada a cada turno aos clientes.
- **WINNER**: Mensagem enviada pelo servidor para informar aos clientes quem venceu a partida e permitir que o vencedor escolha uma carta do deck.
- **ENCERRAR**: Mensagem enviada pelo cliente vencedor para encerrar o jogo, que é repassada a todos os clientes.

A estrutura adotada, apesar de sua complexidade, foi considerada mais adequada para o propósito do projeto, além de mais eficiente uma vez que permite uma comunicação eficiente e distribuída entre os clientes e o servidor.

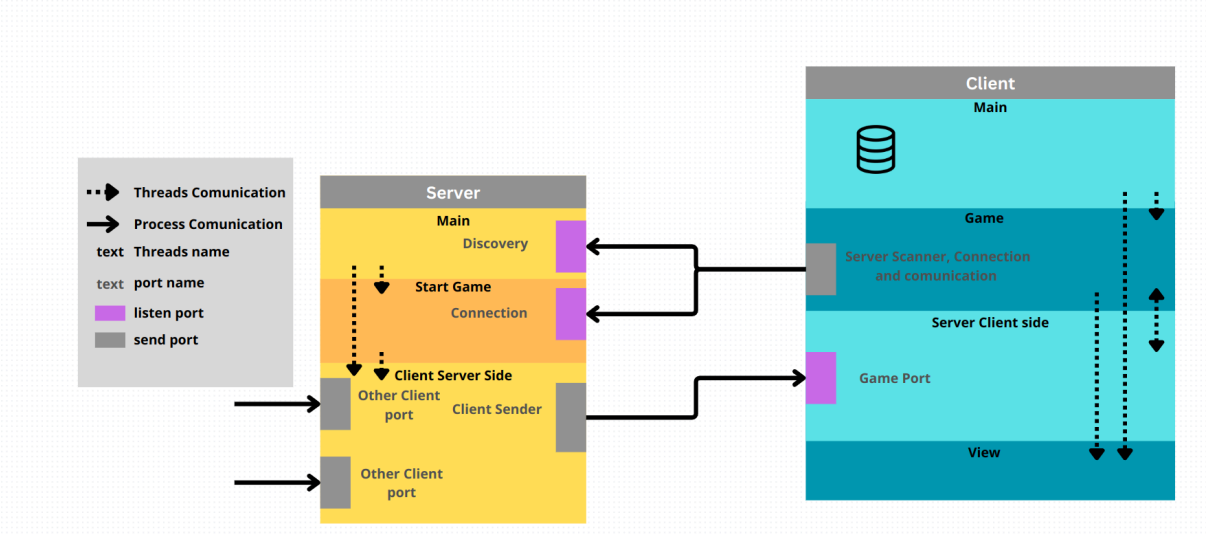

Figura 0 - Arquitetura do projeto

2.4 Interface gráfica (GUI)

No início do desenvolvimento, foi decidido utilizar a biblioteca **Tkinter** para a construção da interface gráfica, nos menus devido à sua simplicidade e facilidade de uso. Essa escolha permitiu que o foco inicial estivesse na implementação das funcionalidades básicas, adotando uma abordagem direta e menos modular. Embora a interface criada com Tkinter fosse funcional, seu design visual era simples e rudimentar, mas atendia adequadamente aos requisitos do projeto.

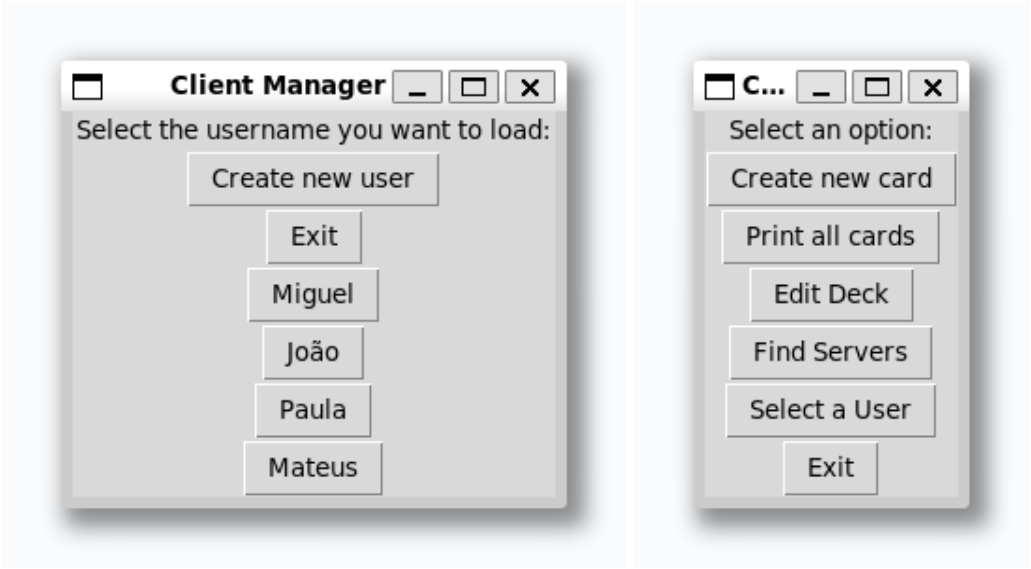

Figura 01 - Telas de menu utilizando o Tkinter

Figura 2 - Imagem da tela de conexão ao servidor feita no Tkinter, para o momento em que os jogadores ainda estão se conectando

Com o objetivo de oferecer uma interface mais robusta e visualmente atraente, optamos por migrar toda a interface para o **Pygame**. Embora o Pygame seja uma biblioteca mais poderosa em termos de personalização gráfica, ele não oferece componentes de interface prontos como o Tkinter, o que torna sua utilização significativamente mais complexa.

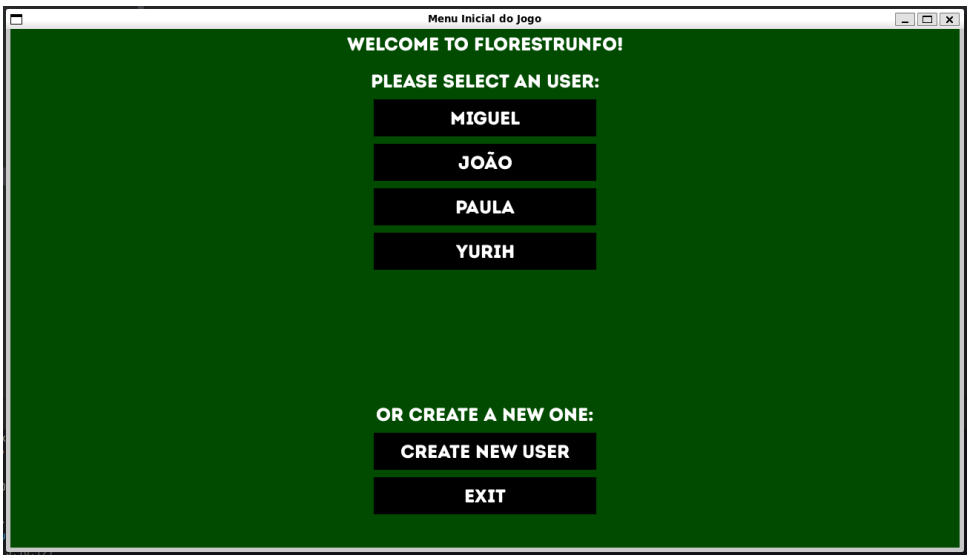

Figura 3 - Tela de menu inicial feita no Pygame

No **Pygame**, tornou-se necessário desenvolver manualmente classes para os principais componentes da interface, como botões **Button**, caixas de texto **TextBox** e cartas selecionáveis **CardSelector**. Cada componente foi encapsulado em sua própria classe, o que facilitou a manutenção e a reutilização do código.

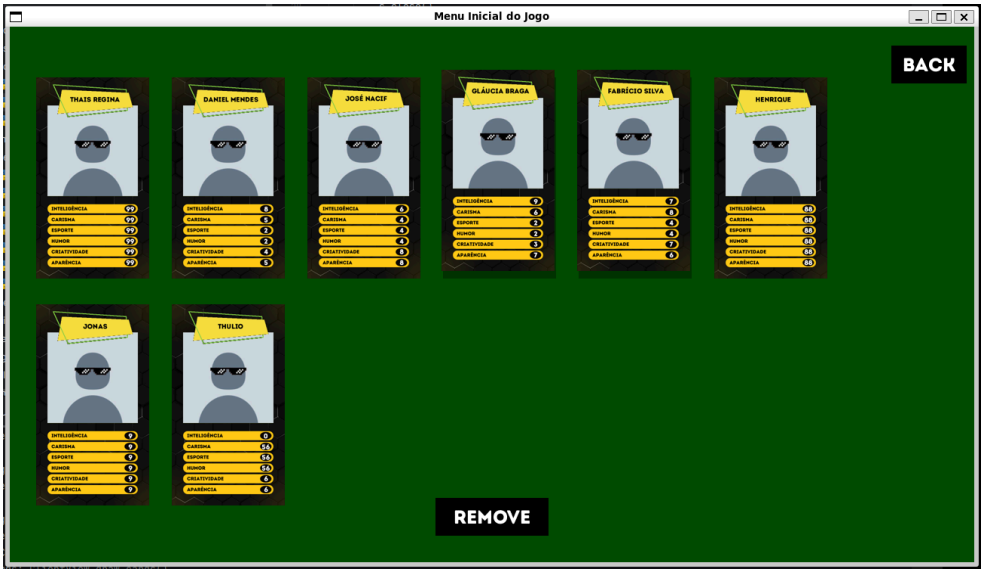

Figura 4 - Tela de remoção de cartas no deck, com as cartas de Gláucia e Fabrício selecionadas

O maior desafio durante a migração para o **Pygame** foi integrar a nova interface modularizada com o código existente, que estava fortemente acoplada à interface do **Tkinter**. Infelizmente, a implementação inicial com Tkinter não seguiu uma abordagem modular, resultando em uma base de código onde a lógica da interface gráfica estava dispersa por todo o projeto, criando dependências difíceis de resolver.

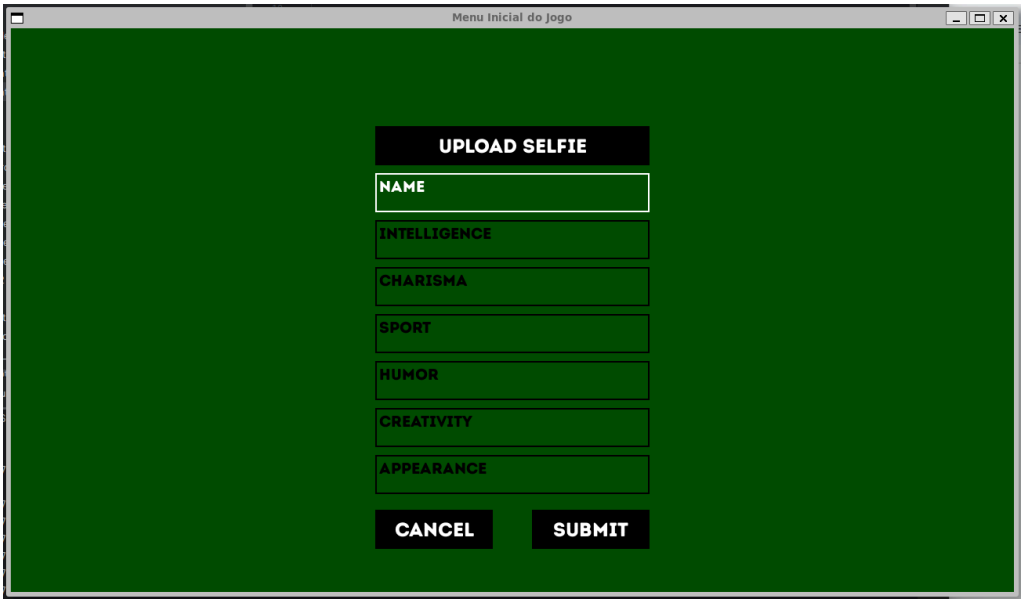

Figura 5 - Tela de cadastro de nova carta

Por fim, decidimos nessa parte não entregarmos a interface no PyGame finalizada e sim pelo **Tkinter**, para não prejudicar o que é mais importante, a implementação pela **API** de **Sockets**. Algumas outras telas interessantes:

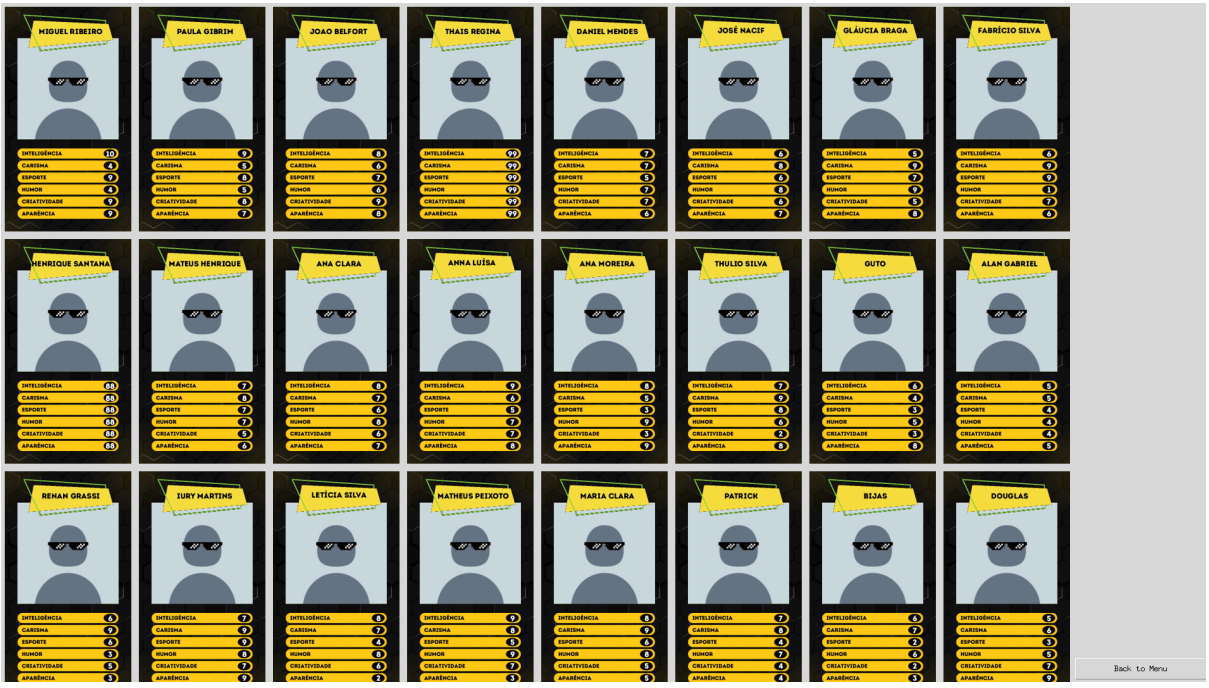

Figura 6 - Cartas do usuário

Figura 7 - O jogo

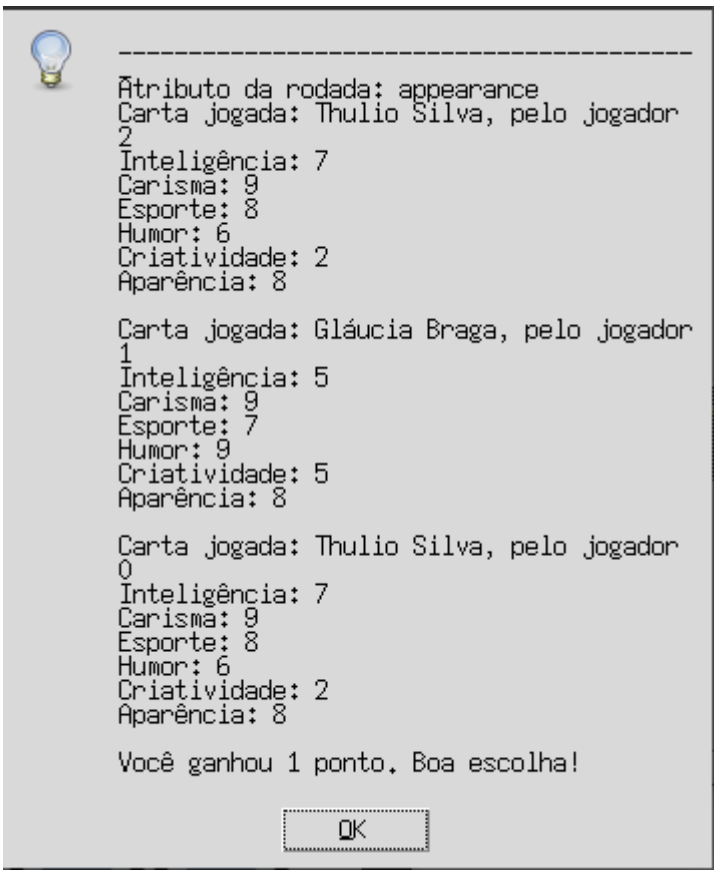

Figura 7 - Log do turno

3. Conclusão

Optamos por utilizar o **Makefile** em vez do Docker devido à sua simplicidade em comparação com a complexidade do **Docker**, que apresentava erros que não conseguimos corrigir a tempo. O trabalho está no github: https://github.com/VicJoao/UFV-Trunfo

Sobre o Banco de Dados: Infelizmente, devido ao tempo limitado, não conseguimos implementar a funcionalidade de **fotos** para as cartas. Reconhecemos que a ausência de fotos é uma limitação, mas enfrentamos dificuldades e a complexidade de integrar esse recurso na interface, especialmente devido ao acoplamento excessivo com o Tkinter. Embora essa funcionalidade seja importante, decidimos adiar sua implementação para focar nas partes principais do projeto.

Sobre a Lógica do Jogo: A lógica atual não trata de algumas questões importantes, como limites para o número de cartas ou a repetição de cartas. Esses aspectos são importantes para um jogo balanceado, mas o foco principal do nosso trabalho não era resolver essas questões detalhadamente. Optamos por permitir maior liberdade ao usuário.

Sobre a Arquitetura: A estrutura adotada, apesar de sua complexidade, foi escolhida. Inicialmente, consideramos a implementação de um sistema cliente-servidor simples, mas decidimos pela arquitetura híbrida cliente-servidor com peer-to-peer (P2P). Essa escolha foi motivada pelo desejo de explorar profundamente o desenvolvimento de sockets, threads e comunicação de mensagens.

Embora tenhamos alcançado um sucesso considerável com essa abordagem, a implementação foi desafiadora. A gestão das diversas estruturas, listas e dicionários para coordenar entre as diferentes threads exigiu um controle rigoroso para evitar confusões. Esse desafio proporcionou uma compreensão mais profunda das técnicas de comunicação distribuída e processamento paralelo, permitindo-nos aprimorar nossas habilidades nessas áreas.

Em resumo, apesar das dificuldades encontradas e das limitações atuais, o projeto avançou significativamente. A arquitetura híbrida e as decisões tomadas contribuíram para um aprendizado valioso.

4. Referências

Slides da disciplina
Canva
How to exit the entire application from a Python thread? - Stack Overflow
List of IP addresses/hostnames from local network in Python - Stack Overflow