

Trabalho Prático 4

Miguel Antônio Ribeiro e Silva - 4680

João Victor Graciano Belfort de Andrade - 4694

Paula Teresa Mota Gibrim - 4234

1. Introdução

O trabalho iniciou-se com um estudo sobre o que é um middleware RMI e como funciona, focando obviamente no Pyro 5, montamos uma implementação básica para simular a comunicação entre clientes.

Para o nosso projeto, realizamos uma análise detalhada e identificamos as áreas que precisavam de mudança. Concluímos que, em grande parte, as modificações foram necessárias apenas nas seguintes partes:

- **Conexão dos Clientes:** Ajustamos a maneira como os clientes se conectam ao servidor.
- **Conexão dos Servidores:** Reformulamos a lógica de conexão do servidor.
- **Troca de Mensagens:** Mudamos o sistema de troca de mensagens
- **Lógica de Interação com o Usuário:** Ajustamos alguns aspectos da lógica que controla a interação entre o jogo e o usuário.
- **Interface e banco de dados:** melhoramos a interface e populamos o banco de dados

No entanto, a lógica central do jogo permaneceu inalterada, pois já atendia às necessidades do projeto. A visualização sofreu pequenas modificações, que detalharei a seguir.

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. Além de iniciar o name server antes de tudo com **pyro5-ns**.

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 **pyro_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 **pyro_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 usando Pyro

2.1 Arquitetura do projeto

A arquitetura do projeto se manteve a mesma, continuamos com um sistema híbrido entre **cliente servidor** com **peer-to-peer** onde os clientes têm a capacidade de interagir ‘diretamente’ e processar dados. Descrevemos como uma arquitetura cliente-servidor com processamento distribuído ou **cliente-servidor híbrido**.

A comunicação continuou **bidirecional**, obviamente, mas para essa implementação não tivemos tempo de dedicar a testes para que ele operasse em LAN, portanto aconselhamos que execute o projeto inteiro em uma máquina local.

2.2 Iniciando o servidor

Para a implementação com **sockets** a lógica era muito focada em portas de conexão, onde o servidor iniciava e esperava os clientes conectarem. Com o **Pyro5** essa lógica foi totalmente reformulada. Para ‘ligar’ um servidor, os seguintes passos foram feitos.

1. Instanciando o Servidor:

```
server = GameServer()
```

- Aqui, um objeto **GameServer** é criado. **GameServer** é uma classe (presumivelmente definida anteriormente no código) que encapsula a lógica do servidor de jogo. Este objeto será registrado com o daemon Pyro5, permitindo que métodos do servidor sejam chamados remotamente por clientes.

2. Criando o Daemon do Pyro5:

- O **daemon** é o núcleo do Pyro5 que gerencia as conexões de rede e trata as chamadas de procedimento remoto (RPC). Ele escuta por solicitações de clientes e despacha essas solicitações para os métodos do objeto registrado (neste caso, **server**).

3. Registrando o Servidor no Daemon:

- O objeto **server** é registrado no daemon. Isso retorna um **URI (Uniform Resource Identifier)** exclusivo, que identifica o objeto **server** na rede. Esse URI é o endereço pelo qual os clientes poderão localizar e interagir com o servidor.

4. Localizando o Nameserver:

```
ns = Pyro5.api.locate_ns()
```

- O código localiza o **Nameserver** do Pyro5, que é um serviço de diretório centralizado usado para registrar e localizar objetos distribuídos pelo nome. Este é um passo essencial para que os clientes possam encontrar o servidor pelo nome em vez de um URI específico.

5. Registrando o Servidor no Nameserver:

```
ns.register("Server", uri)
```

- O servidor é registrado no Nameserver com o nome "**Server**", associado ao URI obtido anteriormente. Isso permite que qualquer cliente que se conecte ao Nameserver procure pelo nome "**Server**" e obtenha o URI para se comunicar com o servidor de jogo.

6. Iniciando o Loop de Requisições:

- **daemon.requestLoop()** é chamado, o que coloca o daemon em um loop infinito, onde ele fica ouvindo por requisições de clientes e respondendo a essas requisições.

2.3 Iniciando os clientes

Na nossa lógica os clientes também conectam de forma bem parecida com os servidores, pois eles também atuam como ‘servidores’ ao processar o jogo, veja:

1. Criando o Daemon do Pyro 5 para o cliente:

- Um daemon Pyro5 é criado para o cliente. Este daemon gerencia as conexões de rede e lida com as chamadas remotas que são direcionadas ao cliente.

2. Registrando o Cliente no Daemon:

- O próprio cliente (**self**) é registrado no daemon. Isso retorna um URI exclusivo que identifica o cliente na rede.

3. Localizando o Nameserver:

- O código localiza o **Nameserver** do Pyro5, que é um serviço de diretório centralizado usado para registrar e localizar objetos distribuídos pelo nome. Este passo é necessário para registrar o cliente com um nome específico, facilitando a localização pelo servidor ou outros clientes.

4. Gerando um ID para o Cliente e Registrando no Nameserver:

- Um ID único é gerado para o cliente usando **random.randint(1, 1000)**. Este ID é então concatenado com o prefixo "**Cliente**" para criar um nome único para o cliente, como "**Cliente123**". Em seguida, o cliente é registrado no Nameserver com esse nome, associando-o ao URI gerado anteriormente.

5. Iniciando o Loop de Requisições em uma Thread Separada:

- O loop de requisições do daemon é iniciado dentro de uma função **run_daemon**, que é passada como alvo para uma nova thread. A thread é iniciada em modo daemon (**daemon=True**), o que significa que ela será executada em segundo plano e será automaticamente finalizada quando o programa principal terminar. O loop de requisições (**daemon.requestLoop()**) permite que o cliente fique escutando chamadas remotas de outros clientes ou do servidor.

2.4 Lógica do servidor e clientes para troca de mensagens

Na implementação original usando sockets, o servidor armazena os IPs e portas dos clientes para enviar e receber mensagens. O servidor também escutava e enviava mensagens utilizando APIs de sockets e portas, com as mensagens encapsuladas em um cabeçalho chamado **Message**, que estava contido na classe **Message**.

Com a transição para Pyro5, essa lógica foi reformulada. Ao iniciar os três clientes e o servidor, duas threads são criadas na classe **Server**.

1. **Thread de Verificação de Jogadores:** Essa thread verifica a quantidade de jogadores conectados a cada segundo, aguardando até que três clientes estejam prontos para iniciar o jogo.

2. **Thread de Verificação de Turnos:** A outra thread verifica a quantidade de jogadas realizadas em cada turno. Quando todos os três jogadores fizerem suas jogadas, o round é processado.

No cliente, uma thread adicional é criada para executar o daemon Pyro5, mantendo-o em execução e escutando chamadas remotas sem bloquear a execução do programa principal.

1. Conectando clientes ao servidor

Uma vez que os clientes estejam carregados e registrados no Nameserver sob o nome "Cliente123", sendo 123 um ID entre 0 e 999, ao clicar em "Escanear servidor", o cliente encontrará todos os servidores conectados ao Nameserver. No entanto, para nossa implementação, restringimos a interface do cliente para mostrar apenas o servidor com o nome "Server" no Nameserver.

2. Confirmando conexão

Após confirmar a conexão, cada cliente envia um ping ao servidor. Se o servidor receber o 'ping', ele responde com um 'pong', indicando que a conexão foi estabelecida com sucesso. Nesse ponto, o cliente pode enviar seus dados de jogo, como o deck de cartas, nome e o ID no Nameserver (chamado de **pyroname**), utilizando um método remoto **ping** no servidor.

3. Enviando os dados para o server

Depois de confirmar a conexão, o cliente envia seus dados ao servidor. O Pyro5 não aceita listas de objetos complexos como o deck de cartas diretamente, exigindo que esses objetos sejam serializados antes do envio. Para isso, implementamos a lógica de serialização na classe **Deck**, que é uma coleção de objetos **Carta**.

Serialização é o processo de transformar um objeto em um formato que pode ser facilmente armazenado ou transmitido, e depois reconstruído. No contexto do Pyro5, esse processo é fundamental, pois o Pyro5 exige que os dados transmitidos entre o servidor e os clientes sejam serializáveis. Isso se torna especialmente relevante quando trabalhamos com objetos complexos, como um deck de cartas, que é uma coleção de objetos **Carta**.

4. Recebimento dos dados pelo servidor

O servidor armazena os dados do jogador e os adiciona à partida, além de incrementar o número de jogadas. Essa lógica é similar à implementação anterior com sockets. No entanto, diferentemente da versão com sockets, nesta implementação Pyro5, o servidor não notifica automaticamente todos os outros clientes quando um novo jogador entra na partida. Embora essa funcionalidade pudesse ser implementada com uma **thread** adicional, decidimos não incluí-la para evitar complexidade desnecessária.

5. Início do jogo

Após enviar seus dados, os clientes aguardam a confirmação do início do jogo pelo servidor. O servidor inicia o jogo assim que a thread de verificação confirma que três jogadores estão conectados e todos enviaram seus dados. O servidor então compacta os dados de todos os jogadores em uma estrutura chamada **game_data**. Esses dados são serializados e enviados para todos os clientes usando seus respectivos **pyronames**. Cada cliente, com os dados recebidos de todos, inicia o jogo. Uma mudança na interface foi que agora o atributo sorteado aparece na tela, em vez de ser exibido em uma caixa de mensagem.

6. O jogo

A lógica do jogo a partir desse ponto segue a mesma estrutura da implementação anterior com sockets, mas com métodos invocados remotamente entre servidor e clientes. Os clientes enviam suas jogadas, representadas pelas cartas escolhidas, utilizando o método remoto **play_card** do servidor. O servidor armazena a jogada de cada cliente em uma lista de jogadas.

A **thread** que verifica o número de jogadas processa o round assim que as três jogadas são recebidas. O servidor então envia a todos os clientes os resultados do round, que são processados localmente pelo método **receive_round_results** de cada cliente. Para evitar travamentos durante o envio de dados, implementamos essa lógica em uma thread separada, o que solucionou problemas de desempenho.

Esse método no cliente processa os resultados de maneira semelhante à implementação anterior com sockets, com algumas adaptações para lidar com os dados serializados pelo Pyro5. Os clientes determinam o vencedor do round, e o jogo continua repetindo esse ciclo até o final.

7. Fim do jogo

Quando o jogo termina, todos os clientes chamam o método **end_game** no servidor. O servidor encerra a partida e reseta seu estado para o início. Os clientes, por sua vez, retornam ao menu principal. O vencedor pode escolher uma carta para adicionar à sua coleção antes de voltar ao menu.

8. Desconectar do servidor

Adicionamos uma opção para que, caso um cliente saia da partida, todos os jogadores sejam removidos e o servidor cancele a partida.

3. Banco de dados

Desde a última entrega, realizamos algumas atualizações pontuais. Em particular, foi adicionada uma coluna específica para armazenar o caminho relativo da imagem principal da carta, que é posteriormente utilizado na geração da figura da carta, conforme detalhado mais adiante.

Embora a aplicação permita que o usuário crie suas próprias cartas durante a inicialização, optamos por utilizar um formulário no Google Forms para possibilitar maior diversidade de cartas nos testes e na apresentação do projeto. Nesse formulário, os alunos puderam definir os atributos desejados, assim como inserir o nome e a imagem correspondente à sua carta personalizada.

Com as respostas coletadas, realizamos a inserção manual dessas cartas no banco de dados, efetuando pequenas modificações conforme necessário. Além disso, para os usuários previamente criados no sistema (Paula, João, Miguel, Henrique e Thais), foram adicionadas todas as cartas geradas com base nas respostas do formulário. Isso permitiu que os usuários tivessem um conjunto diversificado de cartas disponíveis para testes e demonstrações, aprimorando a dinâmica do projeto.

4. Interface

Como mencionado na documentação anterior, enfrentamos algumas dificuldades ao implementar a interface com pygame conforme planejado. Por isso, para esta entrega, optamos por manter a interface construída com o **Tkinter**. Entretanto, é importante ressaltar que a interface em desenvolvimento com **Pygame** apresentou melhorias significativas, conforme demonstrado nas imagens em anexo.

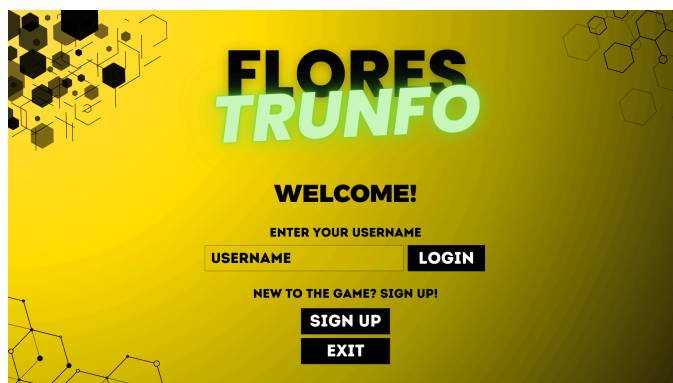


Figura 01 - Nova tela desenvolvida para login de usuários

Em termos de design e experiência do usuário, a interface com **Tkinter** não sofreu grandes alterações desde a última entrega, podendo-se destacar principalmente a opção de upload de imagem para a carta. Contudo, foi detalhado, mais adiante, os aspectos específicos relacionados à criação e visualização das cartas no jogo.

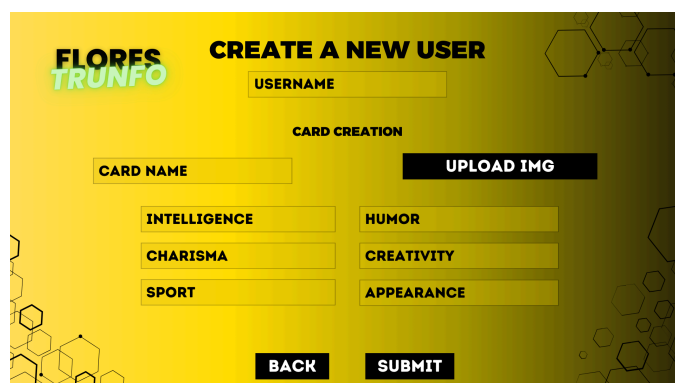


Figura 02 - Nova tela desenvolvida para cadastro de um novo usuário

Essas melhorias refletem nosso esforço contínuo para aprimorar a interface com **Pygame**, que permanece em evolução e deverá substituir a versão atual com **Tkinter** nas próximas iterações do projeto, a nossa intenção é futuramente fazer isso.

O processo de geração de imagens de cartas no jogo é feito no método **gen_card_img** do arquivo **cards.py**, fazendo uso da biblioteca **Pillow** (Python Imaging Library - PIL) para manipulação de imagens, permitindo a construção de cartas visuais com base em templates e atributos fornecidos. A função trabalha diretamente com as imagens do usuário, que são manipuladas para adequar-se ao formato específico da carta.

Inicia-se com a entrada de uma imagem pelo usuário, de tal forma que o arquivo de imagem fornecido servirá como a figura principal na carta. Caso o usuário não forneça uma imagem, será utilizada uma imagem padrão chamada **default.jpg**. O arquivo de imagem é carregado no momento em que a classe **Card** é instanciada. É importante mencionar que entende-se que o usuário não irá incluir uma foto que já consta no diretório padrão de fotografias do projeto, de forma que o sistema automaticamente carrega e cria uma cópia dessa imagem no mesmo.

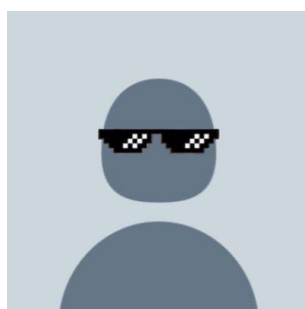


Figura 03 - Imagem escolhida para o padrão.

A função **gen_card_img** então se encarrega de carregar os elementos gráficos da carta, localizados na pasta **elements**. Esses elementos incluem as partes fixas do design da carta, sendo o cabeçalho e o plano de fundo.

Uma vez que o arquivo de imagem do usuário tenha sido carregado, o próximo passo é ajustar o tamanho da imagem para que ela se encaixe perfeitamente na área designada. Essa área está ilustrada no template base da carta, representando uma figura de paisagem no design.

Para assegurar que a imagem do usuário seja cortada corretamente, o algoritmo realiza as seguintes verificações:

1. **Orientação da Imagem:** O código verifica se a imagem está em orientação paisagem (horizontal) ou retrato (vertical).
2. **Ajuste Proporcional:** A imagem é redimensionada de maneira que a menor dimensão (largura ou altura) seja ajustada para 200 pixels. A outra dimensão é ajustada proporcionalmente para manter a integridade da imagem.
3. **Recorte Centralizado:** Após o redimensionamento, a imagem é cortada de maneira a preservar o centro, eliminando as bordas. Isso garante que a imagem final tenha exatamente o tamanho de **200x200 pixels**, que é o espaço reservado na carta para a figura principal.

Esse corte foi planejado para preservar a parte mais importante da imagem, garantindo que o foco principal da foto permaneça visível dentro do template. A partir disso, a imagem é sobreposta ao template base da carta. A imagem manipulada é inserida na área específica do design.

Além disso, o cabeçalho da carta é configurado para permitir a inclusão do nome da carta e outros atributos. A função auxiliar **write_text_on_image** é utilizada para escrever o nome e os atributos da carta nas posições predefinidas do layout.



Figura 04 - Fundo base para as cartas



Figura 05 - Exemplo de carta gerada

Após completar todas as etapas de manipulação, a função **gen_card_img** retorna a imagem gerada da carta, que é usada durante o jogo, servindo como a representação visual de uma carta completa, incluindo os dados personalizados pelo jogador.

Uma dificuldade encontrada diz respeito à partilha de cartas quando o sistema está sendo executado em máquinas diferentes. Isso ocorre pois, até o momento, o banco de dados foi modelado para armazenar um texto com o caminho relativo do arquivo da figura principal daquela carta. Todavia, quando uma outra máquina acessa aquela carta, o arquivo de imagem não existe e, portanto, não funciona conforme desejado. Uma solução para isso seria uma modificação do banco de dados para armazenar a imagem como BLOB (Binary Large Object), que infelizmente não ficou funcional em tempo hábil.

5. Conclusão

Comparando a implementação anterior com sockets e a nova abordagem com Pyro5, podemos observar que a lógica central do jogo permanece a mesma. O que muda é a maneira como as mensagens são enviadas e recebidas. Enquanto a API de sockets exigia um processamento manual e direto das mensagens, o Pyro5 abstrai grande parte dessa complexidade, permitindo a invocação direta de métodos remotos. No entanto, essa mudança trouxe desafios, como a necessidade de serializar dados complexos, o que demandou ajustes na implementação, especialmente na forma como objetos são enviados e recebidos entre cliente e servidor.

Para a visualização decidimos deixá-la pelo Tkinter, infelizmente não tivemos tempo para integrar a interface completamente ao Pygame

6. Referências

Pillow – <https://pillow.readthedocs.io/en/stable/reference/index.html>

Pygame – [Pygame Front Page — pygame v2.6.0 documentation](#)

Pyro 5 – [Pyro5](#)

Tkinter – [tkinter — Python interface to Tcl/Tk](#)