UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

JOÃO PEDRO MISSIAGIA

GUSTAVO VINICIUS CAVALLIN BORGES

DESENVOLVIMENTO DE UM SISTEMA COOPERATIVO MULTIJOGADOR UTILIZANDO RPC COM A BIBLIOTECA RPYC

Sistemas Distribuídos e Tecnologias

SANTA HELENA

2025/1

JOÃO PEDRO MISSIAGIA
GUSTAVO VINICIUS CAVALLIN BORGES

DESENVOLVIMENTO DE UM SISTEMA COOPERATIVO MULTIJOGADOR UTILIZANDO RPC COM A BIBLIOTECA RPYC

Sistemas Distribuídos e Tecnologias

Development of a Cooperative Multiplayer System Using RPC with the RPyC Library

Trabalho de Conclusão de Disciplina de Graduação apresentado como requisito para conclusão da disciplina de Sistemas Distribuídos e Tecnologias do Curso de Bacharelado em Ciência da Computação da Universidade Tecnológica Federal do Paraná.

Docente: Prof. Rafael Keller Tesser

SANTA HELENA

2025/1

LISTA DE FIGURAS

LISTAGEM DE CÓDIGOS FONTE

LISTA DE ABREVIATURAS E SIGLAS

SUMÁRIO

1 INTRODUÇÃO

1.1 Conceito da Biblioteca RPyC

1.1.1 O que é RPC (Remote Procedure Call)

O conceito de *Remote Procedure Call* (RPC) define um modelo de comunicação em sistemas distribuídos no qual um programa pode executar procedimentos ou funções em outro computador de forma transparente, como se estivesse realizando uma chamada local. Esse paradigma permite que múltiplos processos ou aplicações interajam por meio de uma interface bem definida, sem a necessidade de o desenvolvedor lidar diretamente com os detalhes de rede, como sockets, protocolos de transporte ou serialização de dados.

Na prática, o RPC simplifica a troca de informações entre sistemas distintos, promovendo modularidade, reuso de código e escalabilidade — características essenciais em aplicações distribuídas modernas.

1.1.2 O que é o RPyC e como ele se encaixa nesse conceito

O *RPyC (Remote Python Call)* é uma biblioteca Python que implementa o modelo RPC de maneira simples e eficiente, permitindo que um programa Python invoque funções, acesse objetos e manipule variáveis de outro programa Python executando remotamente. Essa comunicação ocorre de forma transparente, utilizando uma arquitetura cliente-servidor, em que o servidor *RPyC* hospeda os objetos e métodos que podem ser acessados remotamente, enquanto o cliente realiza chamadas a esses métodos como se fossem locais.

O *RPyC* abstrai toda a complexidade da comunicação em rede, oferecendo uma API intuitiva baseada em classes e métodos Python. Dessa forma, ele é amplamente utilizado em ambientes acadêmicos, experimentais e de prototipagem de sistemas distribuídos, devido à sua leveza e facilidade de integração.

1.2 Funcionamento do RPyC

1.2.1 Arquitetura Cliente-Servidor

O funcionamento do *RPyC* baseia-se em uma arquitetura cliente-servidor tradicional. O servidor *RPyC* é responsável por expor os métodos e classes que poderão ser invocados remotamente. Ele é executado de forma contínua, aguardando conexões externas através de uma porta TCP definida (geralmente entre 18800 e 18900).

Já o cliente *RPyC* estabelece uma conexão com o servidor utilizando o endereço IP e a porta configurada. Após a conexão, o cliente pode chamar qualquer método prefixado com `exposed_`, receber retornos, atualizar variáveis e até mesmo compartilhar objetos Python entre as duas partes.

Essa comunicação ocorre de forma síncrona, com serialização automática dos dados, e o *RPyC* se encarrega de manter a integridade das chamadas e dos tipos de dados transmitidos.

1.2.2 Mecanismo de Chamada Remota e Comunicação

Internamente, o *RPyC* utiliza o protocolo TCP para troca de mensagens e uma camada própria de serialização de objetos Python. Quando o cliente invoca um método remoto, o *RPyC* envia uma requisição contendo o nome do método e seus parâmetros ao servidor, que, por sua vez, executa a função e retorna o resultado.

No projeto desenvolvido, essa mecânica é explorada na comunicação entre os módulos `cliente.py` e `JogoService.py`. O servidor *RPyC*, hospedado em uma VPS (*Virtual Private Server* - Servidor Virtual Privado) Linux, expõe os métodos responsáveis por gerenciar o jogo, como a adição de jogadores, votação, controle da história e chat em tempo real. Os clientes conectam-se remotamente ao servidor via IP público, acessando a lógica central do jogo sem que seja necessário conhecimento sobre a rede ou os detalhes de implementação.

1.2.3 Segurança e Concorrência

A aplicação foi desenvolvida com foco acadêmico e demonstração dos conceitos de RPC, portanto não implementa autenticação nem controle de permissões no servidor. Por esse motivo, o uso do *RPyC* deve ser restrito a redes seguras e ambientes de teste, evitando a exposição direta à internet.

O servidor utiliza a classe `ThreadedServer` do *RPyC*, que permite múltiplos clientes simultâneos. O estado compartilhado do jogo é centralizado no objeto `MotorJogo`, garantindo a consistência das votações e das mensagens entre os jogadores. Entretanto, por simplicidade, não há mecanismos avançados de sincronização; em versões futuras, recomenda-se o uso de *locks* ou filas de eventos para evitar condições de corrida em sessões com muitos usuários.

1.3 Justificativa da Escolha

1.3.1 Motivos técnicos para escolha do RPyC

A escolha do *RPyC* foi motivada principalmente pela sua facilidade de uso, compatibilidade com Python e pela ausência de configurações complexas. Diferente de outras soluções,

como o gRPC (que requer definição de contratos em arquivos `.proto`) ou o Java RMI (que depende de uma máquina virtual Java), o *RPyC* permite que o projeto seja completamente desenvolvido e testado dentro do ecossistema Python.

Essa simplicidade acelera o processo de desenvolvimento, reduz a curva de aprendizado e torna o código mais acessível a todos os membros da equipe. Além disso, o *RPyC* oferece suporte nativo à comunicação bidirecional e manutenção de estado, o que é essencial para aplicações interativas e cooperativas como o jogo de aventura desenvolvido.

1.3.2 Integração com o projeto e uso em VPS

A biblioteca também se mostrou ideal para a implantação em um servidor VPS, atendendo ao requisito da disciplina de hospedar a aplicação em um ambiente remoto. O servidor *RPyC* é leve e pode ser executado continuamente em segundo plano, escutando conexões externas de clientes que acessam o jogo por meio do endereço IP público da VPS.

Esse modelo de distribuição demonstra, na prática, o funcionamento real de um sistema distribuído: múltiplos clientes, conectados de locais diferentes, interagindo com um servidor remoto que centraliza a lógica e o estado da aplicação. Dessa forma, o uso do *RPyC* permitiu não apenas atender aos requisitos do trabalho, mas também simular um cenário real de comunicação em rede entre processos, consolidando os conceitos estudados em Sistemas Distribuídos.

2 DESENVOLVIMENTO DA APLICAÇÃO

2.1 Visão Geral da Arquitetura

A aplicação foi desenvolvida utilizando a linguagem Python 3.12, adotando o modelo cliente-servidor com comunicação baseada em chamadas remotas via biblioteca *RPyC*. O sistema foi projetado para que múltiplos jogadores possam participar de uma mesma história interativa, discutindo e votando nas decisões por meio de um chat em tempo real.

A arquitetura segue uma estrutura em camadas, inspirada no padrão de separação de responsabilidades. Cada camada possui uma função bem definida, o que facilita a manutenção, os testes e a escalabilidade do sistema.

- **Camada de Lógica (Motor do Jogo):** responsável pelas regras, gerenciamento da história, votos e mensagens do chat.
- **Camada de Serviço (Servidor RPyC):** expõe as funcionalidades do motor do jogo através da rede.
- **Camada de Interface (Cliente):** conecta-se ao servidor, exibe a história e interage com o jogador.

A Figura 1 ilustra a organização geral da aplicação e a relação entre os módulos.

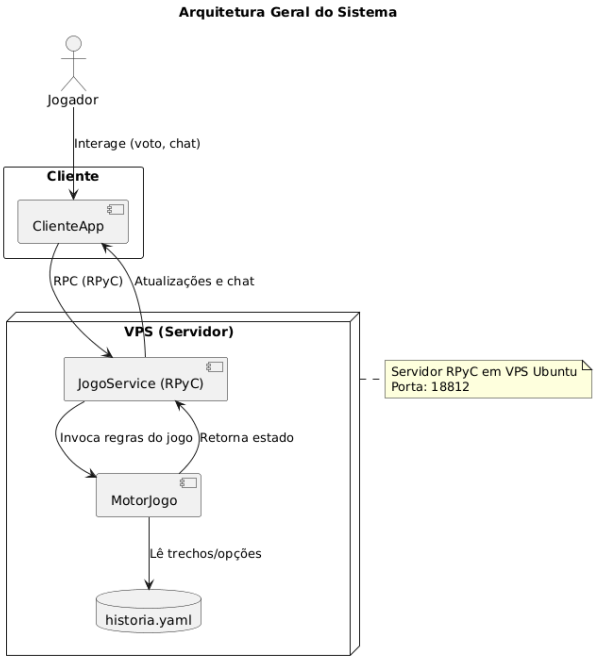

Figura 1 – Arquitetura geral do sistema cliente-servidor.

2.2 Camada de Lógica: Motor do Jogo

A camada central da aplicação é o módulo `motor_jogo.py`, que implementa a classe `MotorJogo`. Essa classe é responsável por controlar o andamento da história, armazenar os votos dos jogadores e gerenciar o chat em tempo real.

A história interativa é carregada a partir de um arquivo no formato *YAML*, o que permite uma estrutura legível e fácil de editar. Cada trecho contém o texto narrativo e as opções de escolha que direcionam o fluxo da história.

Listagem 1 – Estrutura da classe `MotorJogo`.

```
1  class MotorJogo:
2      def __init__(self, arquivo_historia: str): #construtor da classe
3          self.historia = self.carregar_historia(arquivo_historia)
           ↪  #armazena o arquivo yaml da historia
4          self.trecho_atual = None #armazena o trecho atual da historia
5          self.votos = {} #dicionario para armazenar os votos dos
           ↪  jogadores
6          self.chat = [] #lista para armazenar as mensagens do chat
7          self.jogadores_conectados = {}  #dicionario para armazenar os
           ↪  jogadores únicos
8          self.jogo_iniciado = False #flag para verificar se o jogo foi
           ↪  iniciado
9          self.jogadores_prontos = set()  # quem clicou em "Continuar"
10         self.proximo_trecho_pendente = None  # trecho aguardando
           ↪  todos confirmarem
11         self.lock = threading.RLock()
12         self.avancando = False  # flag para impedir confirmações
           ↪  simultâneas
13         self.resultado_calculado = False  # evita calcular mais de
           ↪  uma vez por rodada
14         self.ultimo_resultado = None  # salva o texto do último
           ↪  resultado da votação
```

O método `carregar_historia()` lê o arquivo `historia.yaml` e converte o conteúdo em uma estrutura Python (dicionários e listas), permitindo fácil manipulação durante o jogo. Já os métodos `registrar_voto()`, `enviar_mensagem()` e `obter_trecho_atual()` controlam, respectivamente, o sistema de votação, o chat e o progresso da narrativa.

Listagem 2 – Exemplo de trecho do arquivo `historia.yaml`.

```yaml
acao_direta_e_silenciosa:
  texto: |
    Sem uma palavra, você se torna um redemoinho de ação. Seu ombro
    ↪ atinge Nandlex
    com força total, arremessando-o contra a parede de pedra com um
    ↪ baque surdo. Você
    gira, assumindo uma postura de combate, seus olhos prometendo
    ↪ violência.

    O elemento surpresa é total. Nandlex grunhe, atordoado. Prithen
    ↪ salta para trás, os
    olhos arregalados, pegando a adaga no cinto. O ataque cessou.

    Prithen: Insano! Você morre por esta escória?
    A raiva deles agora está totalmente voltada para você, mas você
    ↪ quebrou o ritmo da
    surra. Orman está salvo por agora, mas a luta de verdade está
    ↪ prestes a começar.
    Mesmo com os Elfos perdendo, mais e mais deles começam a
    ↪ aparecer. Enfim todos
    vocês são derrotados e templários em busca de mendigos para
    ↪ escravizar, acham
    todos vocês que são levados até a senzala.
  opcoes:
    - texto:
      proximo:
```

2.3 Camada de Serviço: Servidor RPyC

O servidor é implementado no módulo `JogoService.py`. Ele atua como uma ponte entre os clientes e o motor do jogo, expondo métodos que podem ser chamados remotamente. Esses métodos seguem o padrão `exposed_nomeDoMetodo`, conforme exigido pela biblioteca RPyC.

Listagem 3 – Serviço RPyC expondo métodos remotos.

```
class JogoService(rpyc.Service):
    def exposed_adicionar_jogador(self, nome):
        return motor_global.adicionar_jogador(nome)

    def exposed_enviar_mensagem(self, jogador, mensagem):
        motor_global.enviar_mensagem(jogador, mensagem)

    def exposed_registrar_voto(self, jogador, opcao):
        motor_global.registrar_voto(jogador, opcao)
```

O servidor é iniciado através do script `servidor_run.py`, que cria uma instância de `ThreadedServer`, escutando na porta `18812`. Esse servidor foi configurado para rodar em uma VPS Linux, tornando o jogo acessível a qualquer cliente remoto via endereço IP público.

Listagem 4 – Inicialização do servidor RPyC.

```
from rpyc.utils.server import ThreadedServer
from service.JogoService import JogoService

print("Iniciando Servidor RPyC...")
t = ThreadedServer(JogoService, port=18812)
t.start()
```

2.4 Camada de Interface: Cliente

O cliente, implementado em `controller_cliente.py`, é responsável por conectar-se ao servidor e fornecer uma interface simples para interação do usuário. Ele permite que o jogador visualize o trecho atual da história, envie mensagens no chat e registre votos nas opções disponíveis.

Listagem 5 – Conexão do cliente ao servidor remoto.

```
import rpyc

class ClienteApp:
    def __init__(self):
        self.conexao = rpyc.connect("IP_DA_VPS", 18812)
        self.servidor = self.conexao.root
        print("[Cliente] Conectado ao servidor RPyC.")
```

Após a conexão, o cliente entra em um loop de atualização que busca periodicamente o estado atual do jogo e do chat. Essa atualização é essencial para que todos os jogadores visualizem as mesmas informações em tempo real.

2.5 Execução e Interação entre os Módulos

O fluxo de execução ocorre da seguinte forma:

1. O servidor é iniciado na VPS e fica escutando conexões externas.
2. Cada jogador executa o cliente localmente e se conecta ao servidor RPyC.
3. Ao conectar-se, o jogador é adicionado à lista de jogadores ativos.
4. Quando quatro jogadores estão conectados, o jogo é automaticamente iniciado.
5. Os jogadores visualizam o trecho atual da história e votam em suas escolhas.
6. O servidor coleta os votos e determina o próximo trecho conforme o resultado.
7. O chat permite a comunicação em tempo real entre os jogadores.

A Figura 2 ilustra o fluxo geral da comunicação entre os componentes.

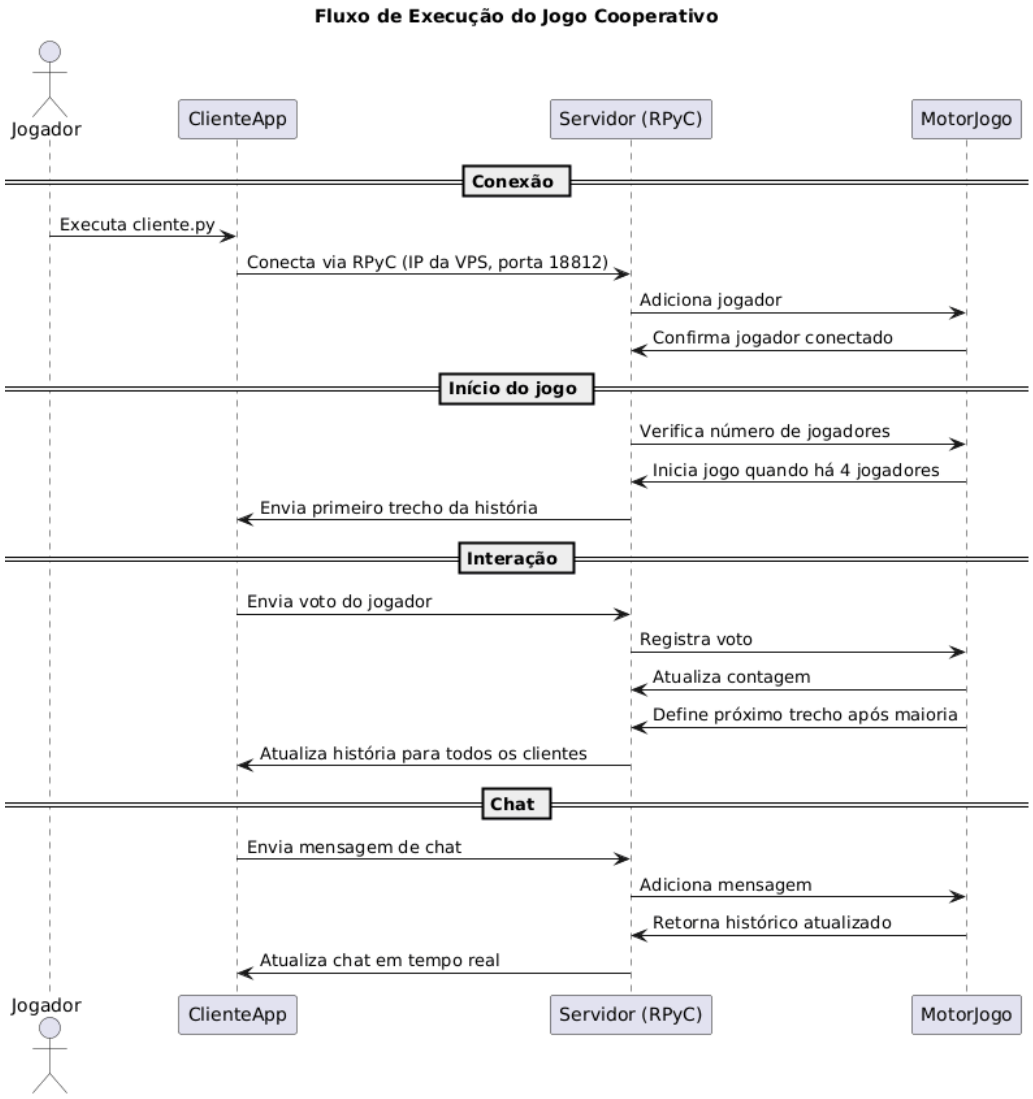

Figura 2 – Fluxo de comunicação entre cliente, servidor e motor do jogo.

2.6 Demonstração e Resultados

Para validação do sistema, o servidor foi inicialmente hospedado de forma local utilizando o ambiente virtual do Python para realização de desenvolvimento e testes antes de realizar a configuração do sistema em um servidor privado virtual (VPS).

Durante os testes, foi possível estabelecer múltiplas conexões simultâneas, realizar votações e interagir pelo chat em tempo real sem falhas de sincronização.

A Figura 3 apresenta a topologia geral de execução, na qual múltiplos clientes se conectam ao servidor RPyC hospedado de forma local. Em seguida, a Figura 4 exibe o terminal do servidor local com os logs de conexão dos jogadores.

A Figura 5 mostra a tela inicial pré-jogo, onde o usuário insere seu nome. Após essa etapa, é carregada uma tela que exibe os jogadores conectados à sessão e quantos ainda faltam, conforme representado na Figura 6. Quando todos os usuários estão conectados, a

história inicial é exibida, como mostra a Figura 7. A Figura 8 apresenta a tela principal do jogo, com a história e suas opções carregadas, além de uma mensagem enviada por outro jogador no chat.

Por fim, a Figura representa a arquitetura do sistema executado de uma VPS.

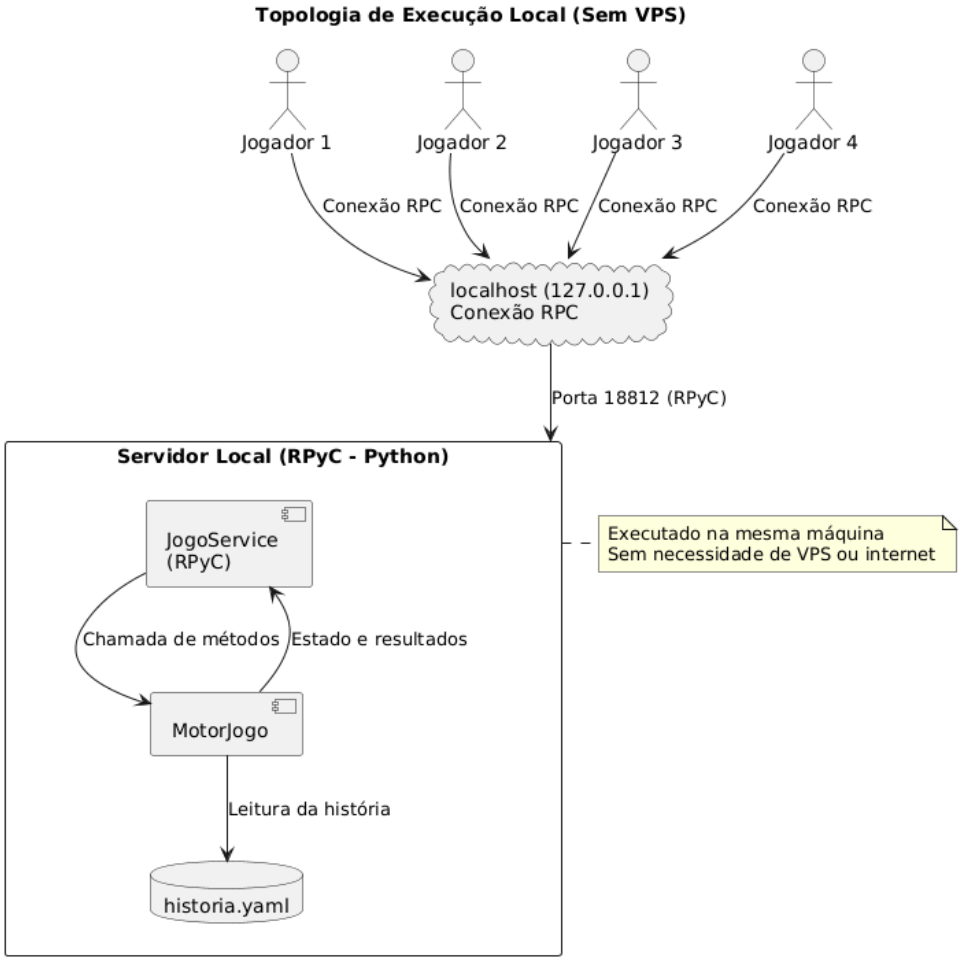

Figura 3 – Topologia de execução: múltiplos clientes conectados ao servidor RPyC hospedado de forma local.

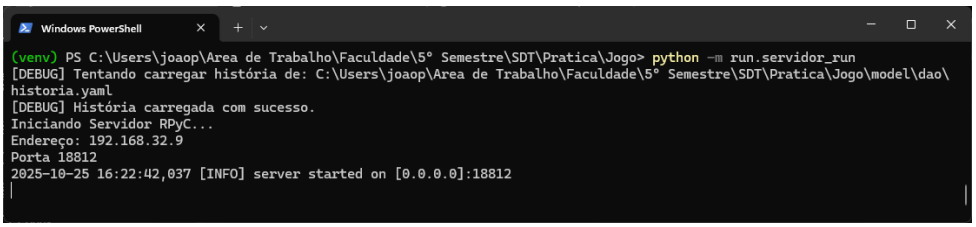

Figura 4 – Servidor RPyC em execução localmente, registrando conexões e eventos.

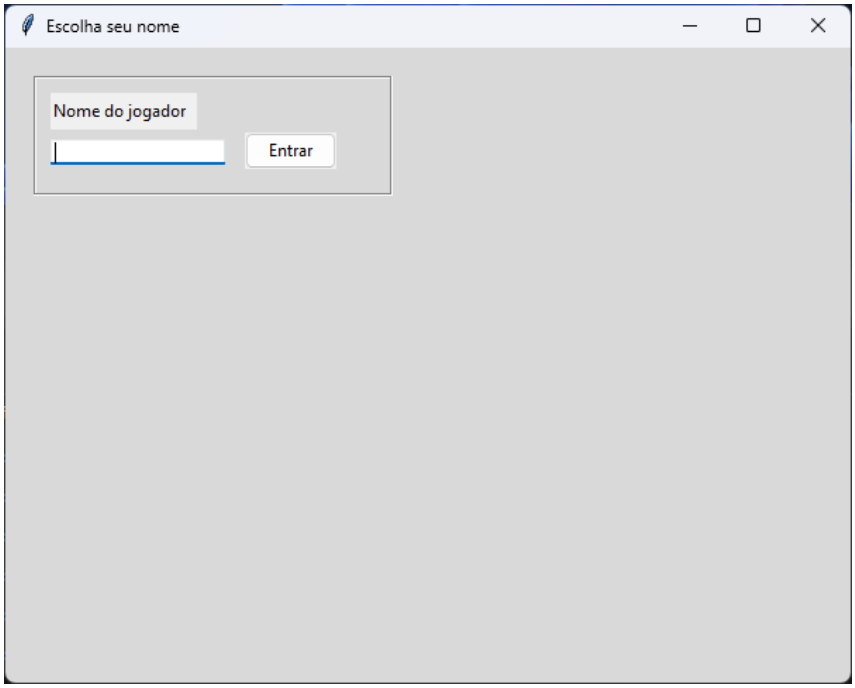

Figura 5 – Tela inicial pré-jogo.

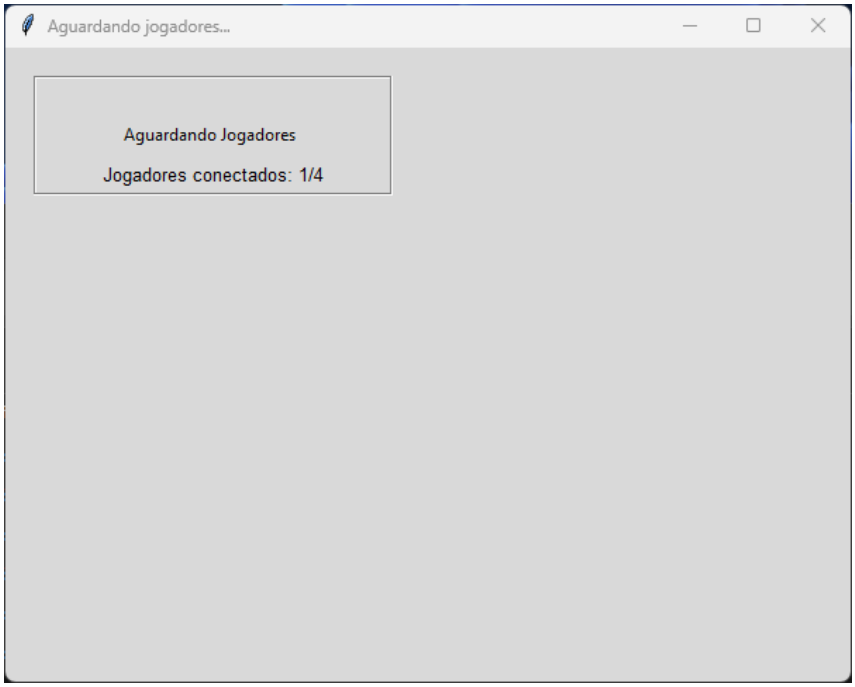

Figura 6 – Tela de espera pelos demais jogadores.

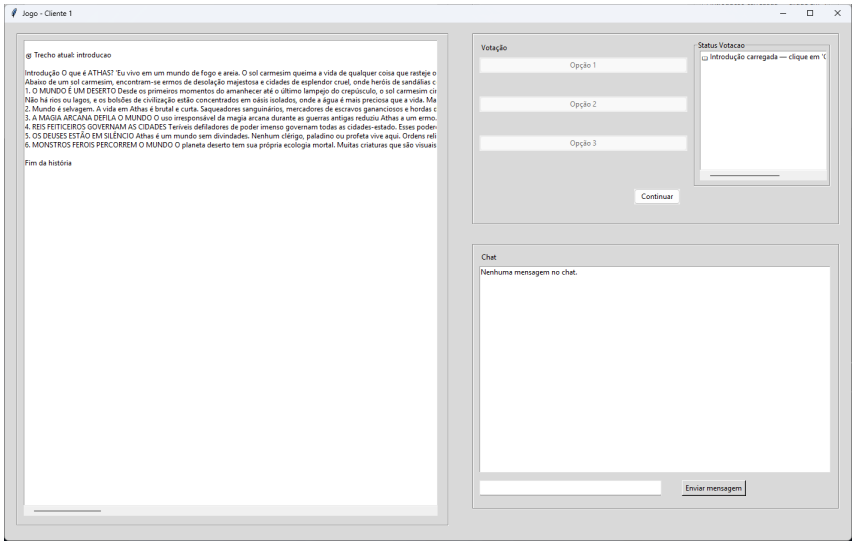

Figura 7 – Tela inicial do jogo após o início da sessão.

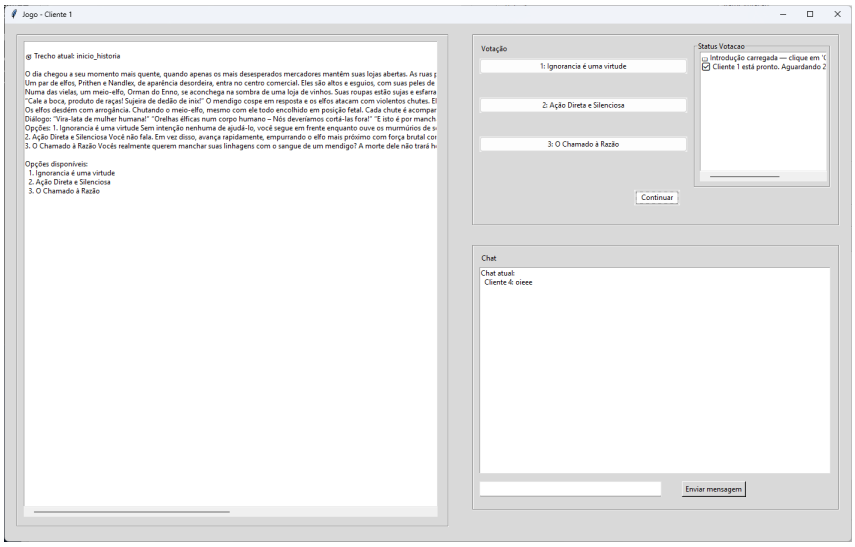

Figura 8 – Tela principal com a história e o chat carregados.

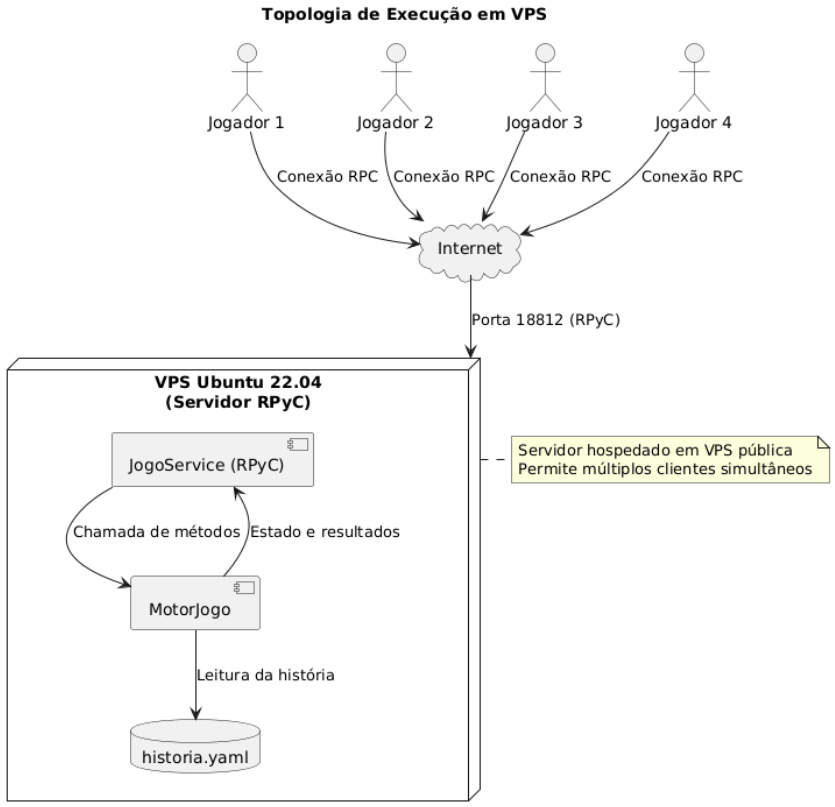

Figura 9 – Topologia de execução: múltiplos clientes conectados ao servidor RPyC hospedado em uma VPS.

2.7 Padrão Arquitetural Utilizado

Durante o desenvolvimento, foi adotado o padrão arquitetural MVC (Model–View–Controller) como base para a organização do código. Esse padrão permite separar claramente as responsabilidades de cada parte do sistema, facilitando a manutenção, os testes e futuras expansões.

- **Model (Modelo):** Representado pela classe `MotorJogo`, que contém toda a lógica e o estado do jogo. É responsável por gerenciar a história, os jogadores, o sistema de votação e o chat.
- **View (Visão):** Implementada no módulo `jogo_interface.py`, responsável por exibir as informações ao jogador e capturar suas ações (mensagens e votos) e os módulos `nome_jogador.py` e `aguardando_jogadores.py` são responsáveis por representarem as interfaces de pré jogo.
- **Controller (Controlador):** Representado pelo serviço `JogoService.py`, que atua como intermediário entre o cliente e o modelo, utilizando a biblioteca RPyC para receber chamadas remotas e repassar as requisições ao motor do jogo.

Além disso, foi adicionada a pasta *controller/*, que centraliza a lógica de controle da aplicação. Essa camada atua como um elo adicional entre o modelo e o serviço de rede, coordenando o fluxo de dados e as interações entre os componentes. Com isso, a estrutura do projeto torna-se mais modular, organizada e condizente com o padrão arquitetural MVC, permitindo maior flexibilidade e facilidade de manutenção.

A adoção dessa arquitetura proporcionou uma clara divisão de responsabilidades entre a lógica do jogo, a comunicação em rede e a interface do usuário. Com isso, o projeto manteve uma estrutura modular e extensível, permitindo a substituição ou evolução de cada componente sem impactar os demais.

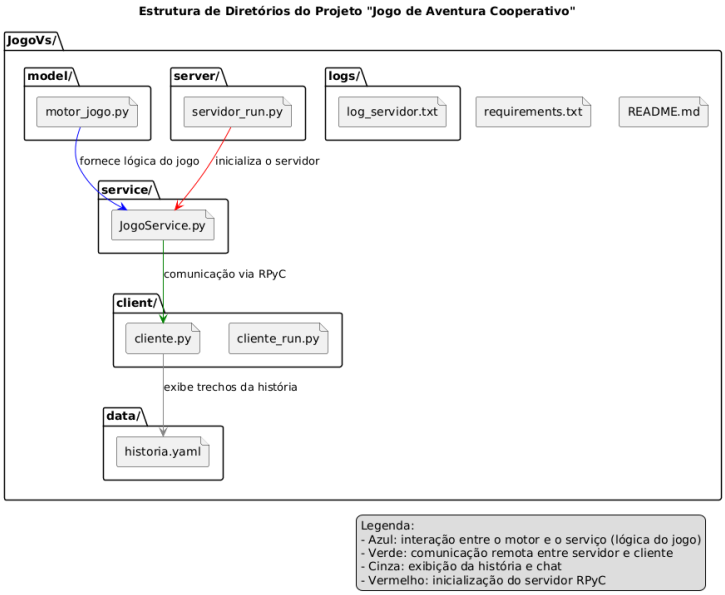

Figura 10 – Estrutura de diretórios e arquivos do projeto, organizada segundo o padrão MVC.

3 INSTRUÇÕES DE INSTALAÇÃO E USO

Este capítulo apresenta o processo completo de instalação e execução da aplicação, cobrindo desde a preparação do ambiente Python até a execução local e distribuída via VPS. As instruções foram elaboradas para que qualquer pessoa, independentemente do nível de experiência, consiga reproduzir o funcionamento do sistema.

3.1 Parte 1: Execução Local (Ambiente de Teste)

Nesta primeira parte, o jogo será executado em um único computador, onde tanto o servidor quanto o cliente rodarão localmente. Esse modo é ideal para testes e verificação inicial do funcionamento da aplicação.

3.1.1 Requisitos

- Sistema operacional: **Windows 10/11** ou **Linux Ubuntu 22.04 ou superior**.
- Python instalado na versão **3.10 ou superior**.
- Editor de código ou terminal (VSCode, PowerShell, Prompt de Comando ou Terminal Linux).
- Conexão à Internet (para instalar dependências).

3.1.2 Verificação do Python

Abra o terminal e execute o comando abaixo para confirmar a instalação do Python:

Listagem 6 – Verificando a instalação do Python.

```
1 python --version
```

A saída deve indicar uma versão igual ou superior a 3.10, por exemplo:

3.1.3 Criação do ambiente virtual

O ambiente virtual (*venv*) é utilizado para isolar as dependências do projeto, evitando conflitos com outras instalações de Python no sistema.

Dentro da pasta do projeto, execute o comando:

Listagem 7 – Criação do ambiente virtual Python.

```
python -m venv venv
```

Em seguida, ative o ambiente virtual: No Windows (PowerShell):

Listagem 8 – Ativando o ambiente virtual no Windows.

```
venv\Scripts\activate
```

No Linux (Ubuntu ou derivados):

Listagem 9 – Ativando o ambiente virtual no Linux.

```
source venv/bin/activate
```

3.1.4 Instalação das dependências

Com o ambiente virtual ativo, todas as bibliotecas necessárias podem ser instaladas automaticamente a partir do arquivo *requirements.txt*, localizado na raiz do projeto. Esse arquivo contém as dependências utilizadas pela aplicação, incluindo as bibliotecas *RPyC* e *PyYAML*.

Listagem 10 – Instalando todas as dependências do projeto.

```
pip install -r requirements.txt
```

Esse comando garante que todas as dependências corretas sejam instaladas de uma só vez, evitando possíveis erros de versão ou pacotes ausentes.

Após a instalação, é recomendável verificar se o pacote foi instalado corretamente:

Listagem 11 – Verificando pacotes instalados.

```
pip list
```

3.1.5 Executando o servidor e o cliente localmente

Com tudo configurado, é possível iniciar o servidor e o cliente na mesma máquina para testes.

1. Abra um terminal e ative o ambiente virtual.
2. Execute o servidor:

Listagem 12 – Iniciando o servidor local.

```
python servidor_run.py
```

3. Em outro terminal, também com o ambiente ativo, execute o cliente:

Listagem 13 – Executando o cliente local.

```
python cliente_run.py
```

Ao conectar, o cliente exibirá a mensagem: [Cliente] Conectado ao servidor RPyC.
E o servidor registrará: [INFO] Novo cliente conectado.
É possível abrir até quatro terminais diferentes executando o cliente simultaneamente para simular uma partida completa.

3.2 Parte 2: Execução Distribuída (Servidor VPS e Clientes Remotos)

Nesta segunda parte, o sistema será executado em ambiente real distribuído, com o servidor rodando em uma VPS Linux e os clientes conectando-se de outros computadores via internet.

3.2.1 Preparando a VPS

3.2.1.0.1 Acesso via SSH

Acesse sua VPS utilizando SSH (substitua pelo seu IP):

Listagem 14 – Acesso remoto à VPS.

```
ssh usuario@IP_DA_VPS
```

3.2.1.0.2 Instalação do Python e Git

Na VPS, atualize os pacotes e instale o Python:

Listagem 15 – Instalação de dependências na VPS.

```
sudo apt update && sudo apt upgrade -y
sudo apt install python3 python3-venv python3-pip git -y
```

3.2.1.0.3 Clonando o projeto na VPS

Listagem 16 – Clonando o repositório do projeto.

```
git clone https://github.com/JaoPdrom/Sistemas-Distrubuidos.git
cd Pratica-1/JogoVs
```

3.2.1.0.4 Criando e ativando o ambiente virtual

Listagem 17 – Criação e ativação do ambiente virtual na VPS.

```
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

3.2.1.0.5 Executando o servidor na VPS

Listagem 18 – Iniciando o servidor RPyC na VPS.

```
python servidor_run.py
```

O servidor exibirá: *Iniciando Servidor RPyC... Servidor ativo na porta 18812*

Para que o servidor continue rodando mesmo se você sair do SSH, pode usar o comando:

Listagem 19 – Mantendo o servidor ativo com o utilitário `screen`.

```
sudo apt install screen -y
screen -S servidor
python servidor_run.py
# Para sair sem encerrar:
Ctrl + A, depois D
```

3.2.2 Configurando acesso remoto

3.2.2.0.1 Verificação do IP público

No terminal da VPS:

Listagem 20 – Obtendo o IP público da VPS.

```
curl ifconfig.me
```

O IP exibido será usado para conexão dos clientes.

3.2.2.0.2 Abrindo a porta 18812 no firewall da VPS

Listagem 21 – Liberando porta para acesso remoto.

```
sudo ufw allow 18812/tcp
sudo ufw reload
```

3.2.3 Executando o cliente remotamente

Nos computadores dos jogadores, siga o mesmo processo da Parte 3.1 até a criação do ambiente virtual e instalação do RPyC.

Em seguida, edite o arquivo *cliente.py* e substitua o IP local pelo IP público da VPS:

Listagem 22 – Configuração do IP do servidor remoto.

```
self.conexao = rpyc.connect("IP_PUBLICO_DA_VPS", 18812)
```

Depois, execute o cliente:

Listagem 23 – Execução do cliente remoto.

```
python cliente_run.py
```

Cada jogador deverá ver a mensagem de conexão bem-sucedida e poderá interagir pelo chat e sistema de votação.

3.3 Conclusão

Seguindo os passos apresentados, é possível instalar, configurar e executar o jogo cooperativo tanto em ambiente local quanto distribuído. A aplicação foi desenvolvida para ser acessível, leve e multiplataforma, garantindo que mesmo usuários iniciantes consigam reproduzir o funcionamento completo do sistema.

4 CONCLUSÃO

O desenvolvimento deste projeto permitiu aplicar, de forma prática, os conceitos estudados na disciplina de Sistemas Distribuídos e Tecnologias, com foco na comunicação entre processos por meio de chamadas remotas (*Remote Procedure Calls* – RPC).

A utilização da biblioteca *RPyC (Remote Python Call)* mostrou-se uma escolha adequada para o objetivo proposto, oferecendo uma interface simples, eficiente e totalmente integrada à linguagem Python. Com ela, foi possível implementar um sistema cooperativo multijogador, em que múltiplos clientes interagem simultaneamente com um servidor remoto através de métodos expostos de maneira transparente.

Durante o processo de desenvolvimento, foram adotadas boas práticas de engenharia de software, como a separação em camadas (lógica, serviço e cliente) e o uso de arquivos externos em formato *YAML* para representar as histórias interativas. Essa estrutura modular facilitou a manutenção, os testes e a expansão do projeto.

Além disso, a execução do servidor em uma VPS Linux demonstrou o caráter distribuído real da aplicação, permitindo conexões remotas e a simulação fiel de um ambiente de rede. O sistema foi validado com múltiplos jogadores, confirmando a estabilidade da comunicação, a sincronização do chat e o funcionamento do sistema de votação.

Como resultado, o projeto atingiu com êxito os objetivos propostos, consolidando os conhecimentos teóricos sobre RPC e sua aplicação prática em sistemas cooperativos. O trabalho também proporcionou uma compreensão mais profunda sobre a arquitetura cliente-servidor e sobre os desafios da comunicação em rede, abrindo caminho para futuras evoluções, como a implementação de interfaces gráficas e persistência de dados.

REFERÊNCIAS

GNU Project. **GNU Screen Manual**. [*S.l.*], 2025. Disponível em: https://www.gnu.org/software/screen/manual/. Acesso em: out. 2025.

PyYAML Project. **PyYAML – YAML parser and emitter for Python**. [*S.l.*], 2025. Documentação oficial. Disponível em: https://pyyaml.org/wiki/PyYAMLDocumentation. Acesso em: out. 2025.

RPyC Developers. **RPyC – Remote Python Call**. [*S.l.*], 2025. Documentação oficial. Disponível em: https://rpyc.readthedocs.io/en/latest/. Acesso em: out. 2025.