Universidade de São Paulo
Escola de Artes, Ciências e Humanidades (EACH)

Exercício Prático 1

Redes de Computadores

Aplicação em simulação de *Batalhas Pokémon*

André Portela Lino – 15634885
Eduardo Almeida Cavalcanti de Melo – 15526004
Higor Gabriel de Freitas – 15575879

São Paulo
17 de Outubro de 2025

Sumário

1 Proposta

O objetivo central deste projeto foi a exploração prática de diferentes aspectos de comunicação e protocolos de rede, sendo o tema do trabalho desenvolver um sistema multiplayer de Batalha Pokémon utilizando uma arquitetura híbrida, que combinasse os pontos fortes de múltiplos paradigmas de comunicação onde fizessem mais sentido lógico. A proposta inicial era criar uma aplicação onde jogadores pudessem se registrar em um servidor central, encontrar oponentes, batalhar em tempo real através de uma conexão direta (P2P) e ter suas estatísticas de vitórias e derrotas salvas de forma persistente novamente no servidor. Ao decorrer do desenvolvimento ocorreram novas ideias que incrementaram ainda mais a complexidade das comunicações, como uma criptografia chave pública e privada, fila de múltiplos desafios simultâneos, além da intercalação de comunicações UDP e TCP. Inclusive, vale ressaltar que muitas vezes optamos por desenvolver diferentes funcionalidades da rede invés de funcionalidades propriamente do jogo, já que esse teria mais relação com engenharia de software, o qual não era nossa proposta. Demos a prioridade do desenvolvimento à exploração de recursos na comunicação, mesmo que não fossem tão fundamentais no nosso contexto, como toda parte de criptografia, que de fato não seria necessária em uma comunicação casual de jogos na prática.

2 Fluxo de Jogo Detalhado

Nesta seção descrevemos um exemplo de passo a passo da experiência de um usuário, além de explicamos sobre o sistema de tipos e fraquezas.

2.1 Registro e Descoberta

1. Ao iniciar o cliente, o usuário se registra no servidor enviando seu nome, portas UDP e P2P, e sua chave pública via TCP.

2. Uma vez online, ele pode solicitar a lista de outros jogadores com o comando `list`.

3. Simultaneamente, o servidor notifica todos os clientes via "broadcast" UDP sempre que um novo jogador entra ou sai, mantendo os jogadores cientes do estado da rede em tempo real. Esse "broadcast" na verdade é manual, IP por IP, já que pensamos em possíveis conexões que atravessariam a NAT.

2.2 Sistema de Desafios e Fila

1. Um Jogador A pode enviar múltiplos desafios para os jogadores B, C e D. Cada desafio é uma mensagem UDP que é enviada diretamente ao oponente.

2. O Jogador B, por sua vez, pode receber desafios de A, C e D, formando uma "fila de aceite"em seu cliente. Ele pode ver os desafios pendentes e escolher qual ele vai querer aceitar ou negar.

3. O primeiro desafio que for aceito por qualquer um dos oponentes de A (B, C ou D) iniciará o processo de batalha para o Jogador A, e os outros desafios pendentes serão ignorados.

2.3 Início da Batalha

1. Antes de enviar ou aceitar um desafio, o jogador escolhe um Pokémon de uma lista aleatória de 10 pokémons, carregados do arquivo `pokemon-data.csv` que possui 918 pokémons.

2. O Pokémon escolhido recebe 4 movimentos aleatórios de sua *pool* de movimentos possíveis, definida em `move-data.csv`.

3. Uma vez que um desafio é aceito,os jogadores terminam de enviar os contextos entre sí (pokemons escolhidos) na fase de preparação, então uma conexão TCP direta (P2P) é estabelecida e a batalha iniciada.

4. O primeiro turno é decidido pela velocidade (*speed*) do Pokémon de cada jogador, ou seja, aquele que tiver o Pokémon mais rápido começa. Em caso de empate, o jogador que iniciou o desafio (*dialer*) tem a prioridade.

5. Durante a batalha, o estado do jogo (HP, etc.) é mantido localmente por ambos os clientes. Apenas as ações (o movimento escolhido em cada turno) são trocadas pela rede, e o resultado é igualmente calculado de forma assíncrona por ambos os lados.

2.4 Sistema de Batalha: Tipos e Fraquezas

Uma das principais funcionalidades do jogo é a implementação do sistema de vantagens e desvantagens de tipo, que afeta diretamente o cálculo de dano. A lógica reside inteiramente no cliente, no módulo `Client/game/battle.py`.

- **Tabela de Tipos (`TYPE_CHART`):** Foi implementado um dicionário que funciona como uma matriz de eficácia. Ele mapeia cada tipo de ataque contra todos os tipos de defesa, retornando um multiplicador (ex: 2.0 para "super efetivo", 0.5 para "não muito efetivo"e 0 para "sem efeito").

- **Cálculo de Dano:** A função `apply_move` utiliza essa tabela para calcular o dano final. O processo é o seguinte:

    1. O dano base é obtido do poder do movimento (`move.power`).
    2. O código consulta a `TYPE_CHART` para obter o multiplicador de eficácia do tipo do ataque contra o `type1` do Pokémon defensor.
    3. Se o defensor possuir um `type2`, o processo se repete e o segundo multiplicador é aplicado sobre o primeiro.

4. O dano final é o poder base do golpe multiplicado por esse fator de eficácia combinado, que é então subtraído do HP do oponente.

3 Especificação da Arquitetura

O sistema foi implementado com uma arquitetura híbrida, utilizando um servidor central para gerenciamento e comunicação P2P para as batalhas, a fim de minimizar a latência e a carga do servidor.

3.1 Arquitetura: Híbrida (Cliente-Servidor + P2P)

- **Modelo Cliente-Servidor:** Utilizado para tarefas que exigem uma autoridade central, como registro de jogadores, matchmaking (encontrar oponentes) e a persistência centralizada de estatísticas. Isso garante que todos os jogadores tenham uma visão consistente dos status de cada jogador.
- **Modelo P2P (Peer-to-Peer):** Utilizado para a comunicação direta e em tempo real durante as batalhas. Uma vez que o servidor conecta dois jogadores, eles estabelecem uma conexão direta, e toda a lógica da batalha (troca de movimentos, cálculo de dano) ocorre sem a intermediação do servidor até o final da batalha.

3.2 Protocolos de Comunicação (TCP e UDP)

A escolha do protocolo de transporte foi feita, em suma, com base nos requisitos de cada tipo de comunicação, contudo, também foi preferível diversificar para podermos aprender o funcionamento de ambas na prática:

- **TCP (Transmission Control Protocol):** Escolhido para todas as interações que exigem alta confiabilidade. É usado para o registro no servidor, para o reporte do resultado da batalha ao final e, crucialmente, para a comunicação P2P durante a batalha, onde a perda de um pacote de movimento quebraria a lógica do jogo.
- **UDP (User Datagram Protocol):** Escolhido para comunicações rápidas e onde a perda de um pacote não é crítica. É usado para o broadcast de eventos do servidor e para todo o sistema de matchmaking, que envolve o envio e a resposta a desafios. Se um pacote de desafio se perder, o usuário pode simplesmente tentar novamente.

3.3 Mensagens do Protocolo

A comunicação é feita através de objetos JSON serializados. As principais mensagens são:

- **Cliente → Servidor (TCP):**
  - {`"cmd": "REGISTER", ...`}: Registra um novo jogador no servidor.
  - {`"cmd": "LIST"`}: Solicita a lista de jogadores online.
  - {`"cmd": "CHALLENGE", "target": "..."`}: Pede ao servidor as informações de rede de um oponente para iniciar um desafio.
  - {`"cmd": "RESULT", ...`}: Enviado pelo vencedor da partida para que o servidor registre o resultado.
- **Servidor → Clientes (UDP Broadcast):**
  - {`"type": "EVENT", "sub": "JOIN", ...`}: Informa que um novo jogador entrou.
  - {`"type": "EVENT", "sub": "LEAVE", ...`}: Informa que um jogador saiu.
- **Cliente ↔ Cliente (Comunicação P2P):**
  - {`"type": "DES", ...`} (UDP): Envia um desafio direto a outro cliente.
  - {`"type": "RES", ...`} (UDP): Responde a um desafio (aceitando ou negando).
  - {`"type": "POKEMON_CHOICE", ...`} (TCP Criptografado): Informa o Pokémon escolhido no início da batalha.
  - {`"type": "MOVE", ...`} (TCP Criptografado): Informa o movimento usado no turno atual da batalha.

3.4 Criptografia P2P

Para garantir a segurança e a integridade das batalhas, a comunicação P2P é totalmente criptografada de ponta-a-ponta, sem que o servidor tenha acesso ao conteúdo. O processo, implementado em `Client/rede/crypto.py`, ocorre da seguinte forma:

1. **Troca de Chaves (Diffie-Hellman):** Ao estabelecer a conexão P2P, os dois clientes trocam suas chaves públicas (baseadas no algoritmo X25519).
2. **Geração de Segredo Compartilhado:** Cada cliente usa sua própria chave privada e a chave pública recebida do oponente para calcular uma chave secreta compartilhada. O algoritmo garante que ambos cheguem à mesma chave sem que ela precise trafegar pela rede.

3. **Criptografia Simétrica (AES-GCM):** Esta chave compartilhada é então usada com o cifrador AES em modo GCM (Galois/Counter Mode) para criptografar e autenticar todas as mensagens subsequentes da batalha (escolha de Pokémon, movimentos, etc.).

4 Organização do Código

O projeto está organizado em duas pastas principais: `Client` e `Server`. Os principais arquivos são:

- **Server/server.py:** Contém toda a lógica do servidor. A função `handle_client` é o ponto de entrada para cada cliente conectado e roda em sua própria thread. Como sua funcionalidade não é tão extensa, não foi realizada a divisão por módulos como feito no cliente.
- **Client/main.py:** É o ponto de entrada principal (*main*) da aplicação cliente. Ele inicializa todos os outros módulos do cliente e executa o loop de comandos do usuário além de conter algumas funções gerais.
- **Client/battle.py:** Encapsula toda a lógica de uma batalha P2P, desde a conexão e troca de chaves até o cálculo de dano e o loop de turnos.
- **Client/comunicacaoServer.py:** Gerencia a comunicação TCP com o servidor.
- **Client/queueManager.py:** Orquestra o sistema de desafios via UDP.
- **Client/crypto.py:** Isola toda a lógica de criptografia (troca de chaves e cifragem das mensagens da batalha).

4.1 Visualizando Mensagens de Depuração (Debug)

Para facilitar a análise do código e a visualização detalhada do fluxo de mensagens trocadas, o nível de log pode ser configurado tanto no cliente quanto no servidor. Por padrão, ele está definido como `INFO` para uma visualização mais limpa, mas pode ser alterado para `DEBUG` para exibir todas as "mensagens brutas"que deixamos no código.

Para habilitar o modo de depuração, basta editar a primeira linha de código após as importações nos arquivos `Client/main.py` e `Server/server.py`.

Comente essa linha (adicionar um # no início dela):

```
logging.basicConfig(level=logging.INFO, format='[%(levelname)s] %(message)s')
```

E descomente esta (remover o # no início dela):

```
logging.basicConfig(level=logging.DEBUG, format='[%(levelname)s] %(message)s')
```

Isso revelará um nível muito mais detalhado de informações sobre a comunicação de rede e o estado interno da aplicação nos terminais, facilitando observação de chaves, mensagens, keepalives etc...

4.2 Gerenciamento de Threads

O sistema utiliza múltiplas threads para evitar bloqueios:

- **Servidor:** Possui a thread principal, uma thread para verificar clientes inativos (`check_inactive_clients`) e uma thread `handle_client` para cada jogador conectado. Esta última gerencia todas as requisições TCP do cliente, incluindo as mensagens de **keepalive**, que são enviadas periodicamente pelo cliente para manter a conexão ativa e evitar um timeout por inatividade.
- **Cliente:** Possui a thread principal que gerencia o menu, uma thread `Leitor` para as entradas do usuário, uma thread para o listener UDP, uma thread para os desafios enviados e uma thread dedicada ao envio das mensagens de **keepalive** a cada 20 segundos (definido arbitrariamente).

5 Compilação e Uso

5.1 Compilação

O projeto foi desenvolvido em Python 3 e não requer compilação na prática. A única dependência externa é a biblioteca `cryptography`, que pode ser instalada facilmente em ambientes Windows ou Linux.

1. Certifique-se de que o Python 3 está instalado em seu sistema.
2. Navegue até a pasta raiz do projeto pelo terminal.
3. Execute o comando a seguir para instalar a dependência:

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

Nenhum ambiente de desenvolvimento (IDE) específico é necessário, e não há caminhos de arquivo fixos (*hardcoded*) no projeto. Além disso, caso não haja necessidade de compilar, foram criados arquivos executáveis tanto para Windows, quanto para Linux.

5.2 Execução

O projeto pode ser executado de três maneiras, dependendo do sistema operacional e se o Python está instalado no ambiente.

5.2.1 Opção 1: A partir do Código-Fonte (Windows/Linux - Requer Python)

Esta é a forma universal. A partir da pasta raiz do projeto:

1. **Servidor:**

   ```
   python src/Server/server.py
   ```

2. **Cliente:**

   ```
   python src/Client/main.py <meu_nome> <ip_server> <porta_server>
   <minha_porta_udp> <minha_porta_tcp>
   ```

   (Alternativamente, os argumentos como Nome, IP e portas serão solicitadas interativamente pelo terminal na ordem caso não especificados).

5.2.2 Opção 2: Executável para Windows

1. **Servidor:** Execute o arquivo `Server/server.exe`.
2. **Cliente:** Abra um terminal (cmd ou PowerShell) e execute o arquivo `main.exe` passando os argumentos:

   ```
   Client\dist\main.exe <argumentos>
   ```

5.2.3 Opção 3: Executável para Linux

A execução no Linux pode ser um pouco mais complexa.

1. **Servidor:** Abra um terminal, navegue até a pasta `Server/` e execute:

   ```
   ./server_linux
   ```

2. **Cliente:** Navegue até a pasta `Client/` e simplesmente execute o arquivo passando seu nome:

   ```
   ./client_linux <argumentos>
   ```

Observação: Em sistemas Linux, pode ser necessário dar permissão de execução aos arquivos primeiro com o comando `chmod +x <nome_do_arquivo>`. Adicionalmente, talvez seja necessário, na pasta raiz, criar uma .venv e instalar a biblioteca Cryptography (Já especifciado no Requirements.txt)

5.3 Utilização

A interface do cliente é baseada em texto. Após iniciar, os seguintes comandos estão disponíveis:

- `list`: Mostra os jogadores online.
- `stats`: Mostra suas vitórias e derrotas.

- `ranking`: Mostra o ranking geral de todos os jogadores.
- `desafiar <nome>`: Envia um convite de batalha para um jogador específico.
- `aleatorio`: Procura um oponente aleatório para batalhar.
- `aceitar <nome>`: Aceita um convite de batalha recebido.
- `negar <nome>`: Recusa um convite.
- `sair`: Desconecta do servidor e fecha o programa.

Adicionalmente, você pode excluir o conteúdo `player_stats.json` no servidor para não armazenar as estatísticas de teste.

5.4 Exemplo de Interação

Abaixo um exemplo de interação. Vale ressaltar que o ip foi o mesmo para ambos jogadores pois para esse exemplo executamos o jogo no mesmo computador. Certas mensagens de DEBUG, como keepalive, foram omitidas por serem constantes e não tão relevantes para o exemplo.
Vale comentar também a ausência do sistema de fraquezas neste exemplo (mensagens falando se um golpe foi efetivo ou não). Isso se deve pois o sistema ainda não havia sido implementado no momento da interação abaixo.

Servidor

```
Comando: python Server/server.py
[SERVER] Estatísticas de jogadores carregadas: ['B', 'A']
[SERVER] Thread de verificação de inatividade iniciada.
[SERVER] TCP na porta 5000
[SERVER] Registrado A com ip 192.168.0.60 p2p_port 7000
[SERVER] Registrado B com ip 192.168.0.60 p2p_port 7001
[DEBUG] Mensagem recebedida {'cmd': 'CHALLENGE', 'target': 'B'}
[DEBUG] Mensagem recebedida {'cmd': 'CHALLENGE', 'target': 'A'}
[DEBUG] Mensagem recebedida {'cmd': 'RESULT', 'me': 'B',
'opponent': 'A', 'winner': 'B'}
[STATS] Estatísticas de B e A atualizadas e salvas.
```

Figura 1: Log do Servidor.

Jogador A (Desafiante)

```
Comando: python Client/main.py A
Uso fácil: python client.py <meu_nome> <ip_server> <porta_server>
<minha_porta_udp> <minha_porta_p2p>
IP do servidor (Vazio para 127.0.0.1)
Porta do servidor (Vazio para 5000)
Porta UDP (Vazio para 5001)
Porta P2P (Vazio para 7000)
[INFO] 918 Pokémon carregados da base de dados.
[INFO] Registrado no servidor com sucesso
[INFO] UDP listener rodando na porta 5001
Digite comando (list, stats, ranking, desafiar <nome>, aleatorio,
aceitar <nome>, negar <nome>, sair):
[INFO] O jogador B entrou no servidor.
Comando: desafiar B
--- Escolha dentre esses Pokémon para a batalha! ---
  1. Deoxys-Speed
  2. Weezing
  3. Silvally
  4. Kabuto
  5. Gumshoos
  6. Moltres
  7. Axew
  8. Solrock
  9. Pidgeotto
  10. Rotom-Frost
Digite o número do Pokémon escolhido:
Comando: 7
Você escolheu Axew!
[INFO] Desafio enviado para B
[INFO] B aceitou. Iniciando batalha.
[INFO] P2P: conectado a ('192.168.0.60', 7001)
[DEBUG] Shared key e troca de Pokémon feitos com sucesso:
b'\xac\xc1d6X6\xdcVL\x0c\xdeG\xb2\xfd\
xdc\xa6\x9f\x19\x81\x8d\xdbV\x98\x88oZ\x82"\xc8\xd5\xb5='
[INFO] === BATALHA: Axew vs Mr. Mime ===
[INFO] Aguardando movimento do oponente...
[INFO] Oponente usou mega kick. Seu HP: 0
[INFO] Resultado da batalha: B
[DEBUG] Eu não sou o vencedor. Não irei reportar o resultado.
```

Figura 2: Terminal do Jogador A

Jogador B (Desafiado)

```
Comando: python Client/main.py B
Uso fácil: python client.py <meu_nome> <ip_server> <porta_server>
<minha_porta_udp> <minha_porta_p2p>
IP do servidor (Vazio para 127.0.0.1)
Porta do servidor (Vazio para 5000)
Porta UDP (Vazio para 5001)5002
Porta P2P (Vazio para 7000)7001
[INFO] 918 Pokémon carregados da base de dados.
[INFO] Registrado no servidor com sucesso
[INFO] UDP listener rodando na porta 5002
Digite comando (list, stats, ranking, desafiar <nome>, aleatorio,
aceitar <nome>, negar <nome>, sair):
[INFO] Desafio recebido de A
Comando: aceitar A
--- Escolha dentre esses Pokémon para a batalha! ---
  1. Exeggutor-Alola
  2. Metagross
  3. Mr. Mime
  4. Glameow
  5. Pumpkaboo-Super
  6. Weepinbell
  7. Gourgeist-Large
  8. Golett
  9. Torterra
  10. Goomy
Digite o número do Pokémon escolhido:
Comando: 3
Você escolheu Mr. Mime!
[DEBUG] Enviado b'{"type": "RES", "opp": "B", "res": "ACE"}' para
192.168.0.60:5001
[INFO] Aceitei desafio de A
[INFO] P2P: conexão aceita ('192.168.0.60', 60400)
[INFO] === BATALHA: Mr. Mime vs Axew ===
Seu turno! Seus movimentos: Dazzling gleam, Foul play, Mega kick,
Zap cannon
Comando: Mega kick
[INFO] Você usou mega kick. HP oponente: 0
[INFO] Resultado da batalha: B
[DEBUG] Eu sou o vencedor. Reportando o resultado ao servidor.
```

Figura 3: Terminal do Jogador B.

6 Testes realizados e resultados

Foram realizados múltiplos testes para validar a robustez e o funcionamento do sistema. Máquinas utilizadas para os testes:

Windows 11 Pro 24H2 (Build 26100.68990) — 64 bit: Computador de Eduardo, possível de reconhecer por meio dos prints com \Users\Du no terminal.

Windows 10 Pro 22H2 (Build 19045.6456) — 64 bits: Computador de Higor, possível de reconhecer por meio dos prints com PS D:\Higor Freitas\no terminal.

Arch Linux (Kernel 6.17.2-zen1-1-zen) - 64 bits: Segundo computador de Higor, foi utilizado apenas no último teste.

- **Teste de Conexão, Registro, List e Stats iniciais:** Jogadores Eduardo e Higor conseguiram se registrar com sucesso no servidor.
  Após Eduardo se conectar, ele utilizou o comando o list, que funcionou corretamente e mostrou apenas as informações referentes ao único jogador ativo no momento (Que era o próprio Eduardo).
  Logo em seguida, Higor se conectou, e utilizou o comando list, que mostrou corretamente que Eduardo e Higor estavam ativos. Além disso, as informações que Higor havia se conectado apareceram para Eduardo.
  Por fim, Eduardo utilizou stats para ver suas vitórias e derrotas, e como essa tinha sido sua primeira vez se conectando (logo nunca havia batalhado antes), foi mostrado corretamente que ele possui 0 vitórias e 0 derrotas.
  Ademais, o servidor comunicou corretamente que Eduardo e Higor haviam se conectado.

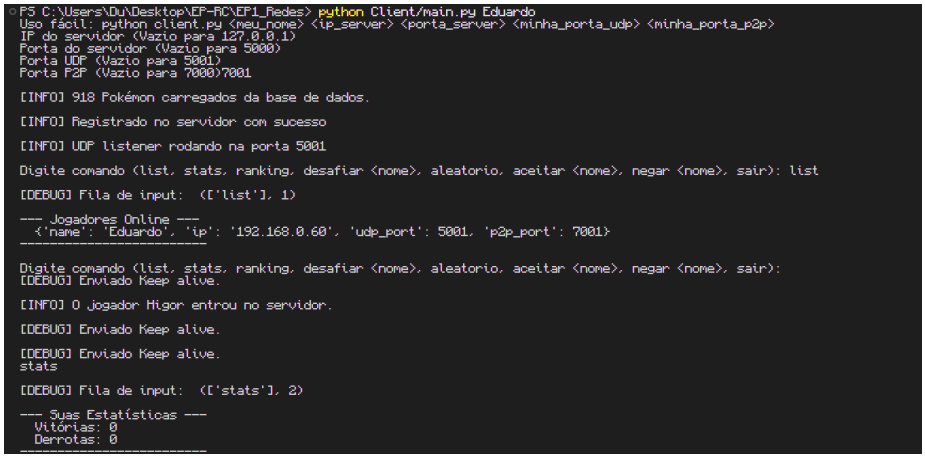

Figura 4: Terminal de Eduardo

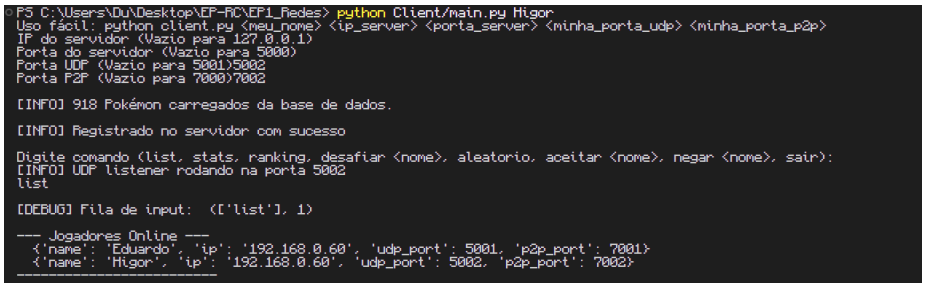

Figura 5: Terminal de Higor

Figura 6: Log do servidor registrando os jogadores Eduardo e Higor

- **Teste de Batalha, Atualização de Stats e de Ranking Geral:** Utilizando os Jogadores Eduardo e Higor, foi realizado o teste de batalha. Primeiramente, Eduardo desafiou Higor. Após isso, 10 pokémons aleatórios foram selecionadas corretamente para que Eduardo escolhesse. Eduardo escolheu o pokémon Gulpin e ficou no aguardo da resposta de Higor.
  Higor corretamente recebeu o desafio de Eduardo, aceitou, apareceu 10 pokémons para que ele selecionasse, ele selecionou Zeraora e dois começaram a batalhar.
  A batalha ocorreu sem nenhum erro ou bug, resultando na vitória de Higor.
  Após a batalha, ambos Eduardo e Higor verificaram seus stats, com ambos stats tendo sido atualizados corretamente. O ranking geral também foi atualizada de forma correta. O log do servidor também mostrou as mensagens e atualizações de forma correta.

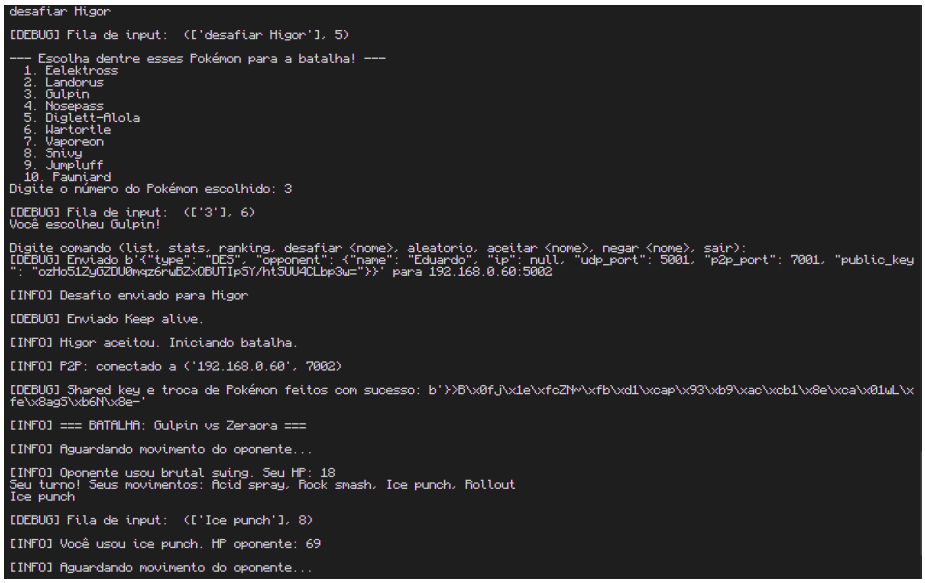

Figura 7: Terminal de Eduardo batalhando - parte 1

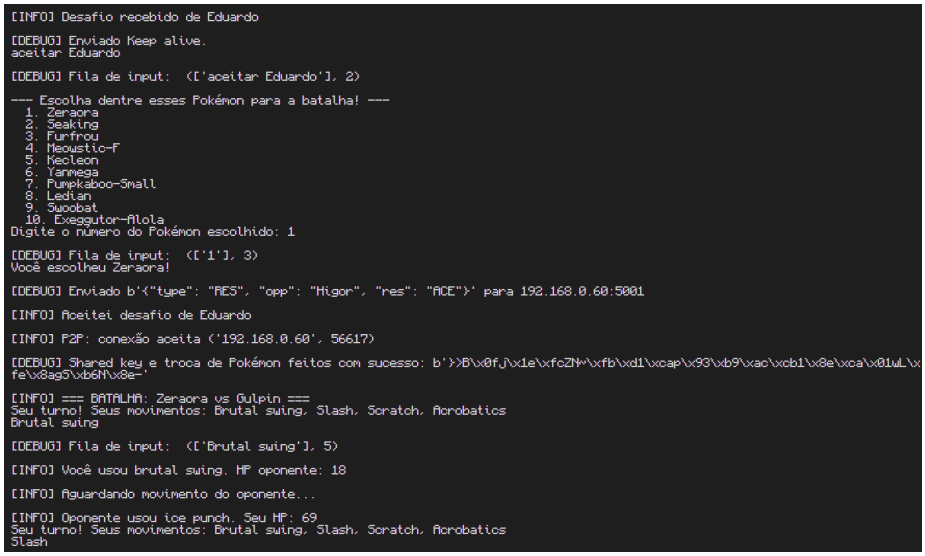

Figura 8: Terminal de Higor batalhando - parte 1

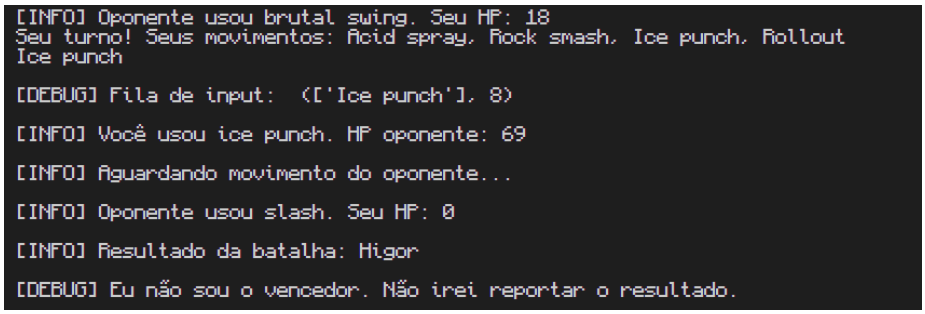

Figura 9: Terminal de Eduardo batalhando - parte 2

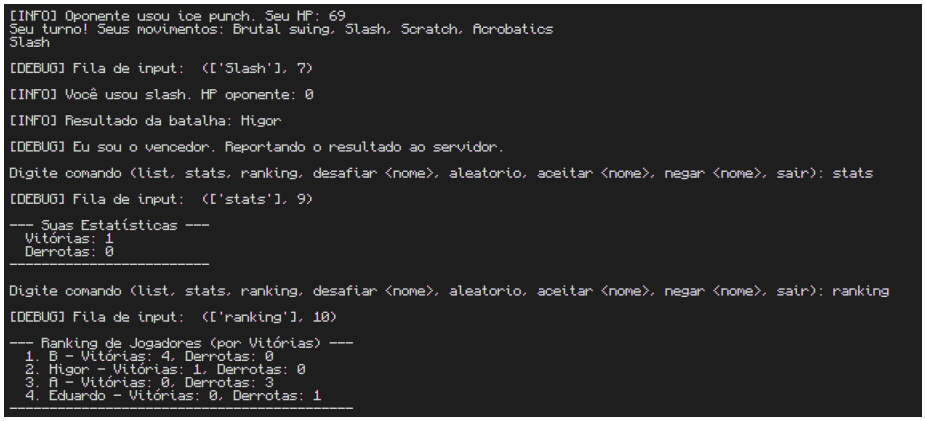

Figura 10: Terminal de Higor batalhando - parte 2

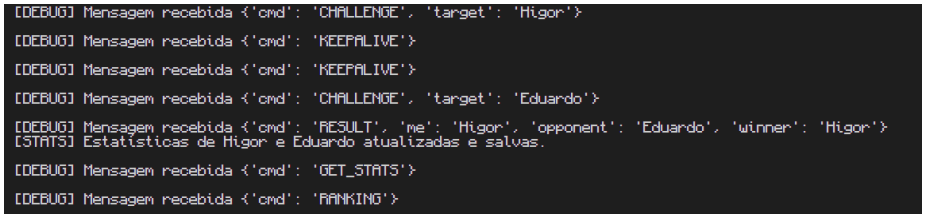

Figura 11: Log do servidor da batalha lendária entre Eduardo e Higor

- **Teste de Comunicação de Saida de Jogador:** Foi feito um teste para verificar se a comunicação de saída estava funcionando. O jogador Higor saiu, servidor recebeu corretamente a mensagem, e salvou as estatistícas de Higor. Eduardo também utilizou list para verificar se apenas ele apareceria na lista de jogadores ativos, o que foi o caso, mostrando que o teste ocorreu com sucesso. **OBS:** É possível perceber que foram carregadas as

estatísticas de alguns jogadores não mostrados anteriormente nos nossos testes. Isso se deve por conta que alguns testes foram feitos em ordens diferentes, por isso que isso ocorre.

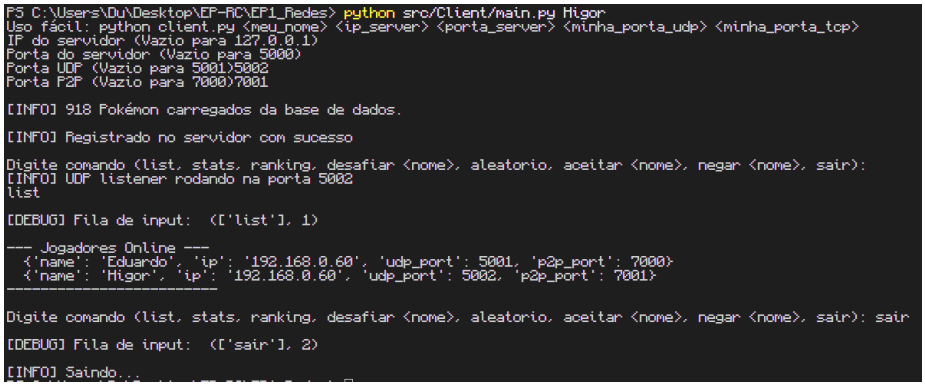

Figura 12: Terminal de Higor ao sair

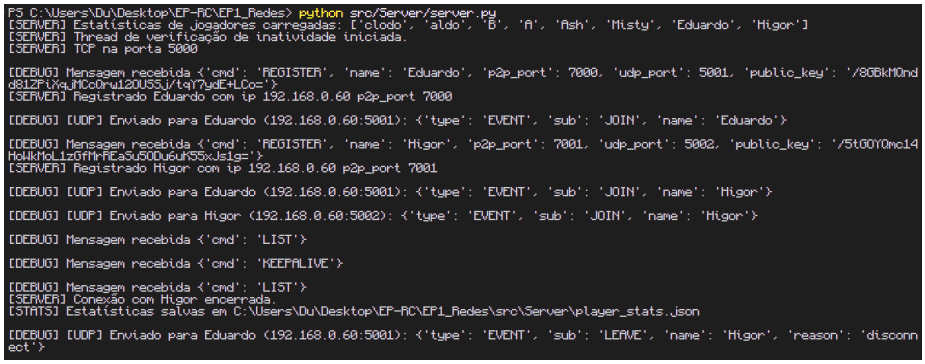

Figura 13: Log do servidor com a saída de Higor

```
PS C:\Users\Du\Desktop\EP-RC\EP1_Redes> python src/Client/main.py Eduardo
Uso fácil: python client.py <meu_nome> <ip_server> <porta_server> <minha_porta_udp> <minha_porta_tcp>
IP do servidor (Vazio para 127.0.0.1)
Porta do servidor (Vazio para 5000)
Porta UDP (Vazio para 5001)
Porta P2P (Vazio para 7000)

[INFO] 918 Pokémon carregados da base de dados.

[INFO] Registrado no servidor com sucesso

[INFO] UDP listener rodando na porta 5001

Digite comando (list, stats, ranking, desafiar <nome>, aleatorio, aceitar <nome>, negar <nome>, sair): list

[DEBUG] Fila de input:  (['list'], 1)

--- Jogadores Online ---
  {'name': 'Eduardo', 'ip': '192.168.0.60', 'udp_port': 5001, 'p2p_port': 7000}
------------------------

Digite comando (list, stats, ranking, desafiar <nome>, aleatorio, aceitar <nome>, negar <nome>, sair):
[INFO] O jogador Higor entrou no servidor.

[DEBUG] Enviado Keep alive.

[INFO] O jogador Higor saiu do servidor.
list

[DEBUG] Fila de input:  (['list'], 2)

--- Jogadores Online ---
  {'name': 'Eduardo', 'ip': '192.168.0.60', 'udp_port': 5001, 'p2p_port': 7000}
------------------------
```

Figura 14: Terminal de Eduardo após a saida de Higor, mostrando que ele era o único jogador ativo no momento

- **Teste de Persistência de Dados:** Ao reiniciar o servidor, as estatísticas foram recarregadas, confirmando a persistência dos dados.

```
PS C:\Users\Du\Desktop\EP-RC\EP1_Redes> python Client/main.py Higor
Uso fácil: python client.py <meu_nome> <ip_server> <porta_server> <minha_porta_udp> <minha_porta_p2p>
IP do servidor (Vazio para 127.0.0.1)
Porta do servidor (Vazio para 5000)
Porta UDP (Vazio para 5001)
Porta P2P (Vazio para 7000)

[INFO] 918 Pokémon carregados da base de dados.

[INFO] Registrado no servidor com sucesso

[INFO] UDP listener rodando na porta 5001

Digite comando (list, stats, ranking, desafiar <nome>, aleatorio, aceitar <nome>, negar <nome>, sair): stats

[DEBUG] Fila de input:  (['stats'], 1)

--- Suas Estatísticas ---
  Vitórias: 1
  Derrotas: 0
-------------------------

Digite comando (list, stats, ranking, desafiar <nome>, aleatorio, aceitar <nome>, negar <nome>, sair): ranking

[DEBUG] Fila de input:  (['ranking'], 2)

--- Ranking de Jogadores (por Vitórias) ---
  1. B - Vitórias: 4, Derrotas: 0
  2. Higor - Vitórias: 1, Derrotas: 0
  3. A - Vitórias: 0, Derrotas: 3
  4. Eduardo - Vitórias: 0, Derrotas: 1
-------------------------------------------
```

Figura 15: Terminal de Higor mostrando que suas estatísticas foram salvas após sua saída e a reinicialização do servidor

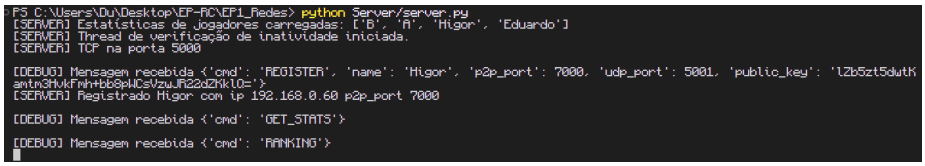

Figura 16: Log do servidor mostrando que as estatistícas atualizadas por conta da batalha entre Eduardo e Higor foram salvas corretamente

- **Teste de Restrição de Auto-Desafio:** Foi realizado um teste simples para validar se a nossa aplicação está impedindo corretamente que um jogador desafie a sim mesmo .
  O jogador Eduardo executou o comando `desafiar Eduardo` para desafiar a si mesmo. O cliente, antes de enviar qualquer pacote de rede para o servidor ou para si mesmo, validou localmente o comando e exibiu a mensagem de aviso "Você não pode se desafiar.", retornando imediatamente ao menu.
  Este teste confirma que a validação de entrada do lado do cliente está funcionando, prevenindo o envio de pacotes desnecessários e a criação de um estado de jogo impossível e sem sentido.

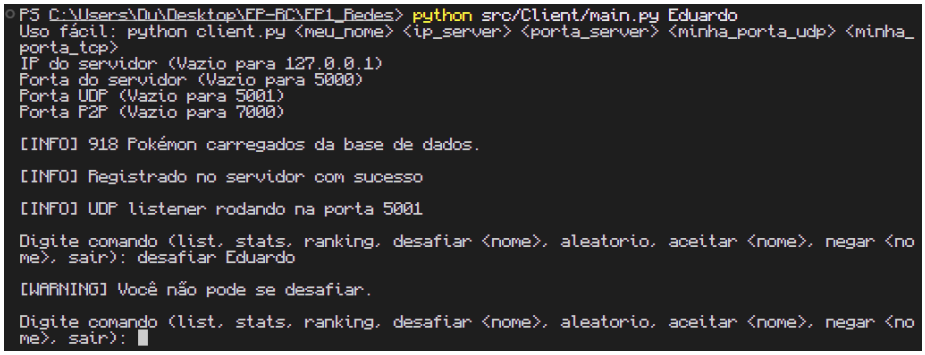

Figura 17: Log do terminal de Eduardo mostrando o impedimento de se auto desafiar e entrar em um estado de jogo impossível.

- **Teste de Conexão com Servidor Offline:** Foi realizado um teste de robustez para verificar o comportamento do cliente ao tentar se conectar enquanto o servidor estava offline.

  O servidor foi mantido desligado e o cliente foi executado. Ao tentar se registrar, o cliente não conseguiu estabelecer a conexão. O sistema tratou a falha corretamente, exibindo a mensagem `[INFO] Não foi possível conectar ao servidor` e `[INFO] Erro, tente colocar um servidor válido` e, em seguida, encerrou o programa de forma controlada, sem travar.

  Adendo: Foi observado também que, em certas condições de falha de

rede, o cliente pode falhar em uma etapa ligeiramente diferente; o menu de comandos pode chegar a aparecer, mas o programa é encerrado imediatamente após a primeira tentativa de comando (como `list`), pois a conexão com o servidor nunca foi estabelecida com sucesso.

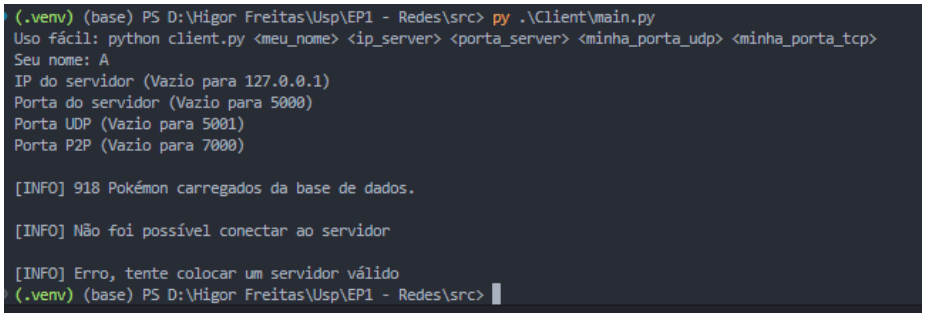

Figura 18: Terminal do cliente mostrando o erro de "Conexão Recusada"ao tentar se registrar com o servidor offline.

- **Teste de Validação de Nome Vazio:** Foi feito um teste de validação de entrada para verificar o que acontecia se o usuário tentasse se registrar com um nome vazio.
  O programa foi iniciado e, ao ser solicitado o nome de usuário, a tecla `Enter` foi pressionada sem que nenhum nome fosse digitado.
  A aplicação validou a entrada localmente, antes de tentar se conectar ao servidor, e exibiu a mensagem de erro de `[ERROR] Falha ao registrar no servidor: 'type'; 'ERR', 'msg': 'missing_fields'`, encerrando a execução em seguida, como esperado. Este teste confirma que a validação de entrada do lado do cliente está funcionando.

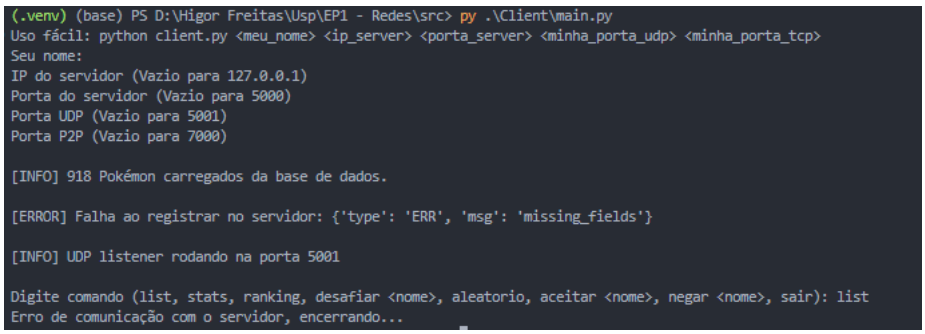

Figura 19: Terminal do cliente mostrando a mensagem de erro ao tentar se registrar com um nome vazio.

- **Teste de Sistemas de Fraquezas e primeiro jogador a agir:** Como foi comentado na seção 5 Exemplo de Interação, e como é possível per-

ceber, nos prints anteriores dos testes não aparece nada a respeito sobre se um golpe foi efetivo ou não (algo comum em jogos de Pokémon). Isso se deve pois inicialmente essa funcionalidade não estava implementada, porém, como o sistema de fraquezas é uma das partes mais legais dos jogos Pokémon, decidimos implementar ela perto do prazo final da entrega do Exercício-Prático.
Também aproveitamos este teste para verificar se o sistema de fazer com que o primeiro jogador a agir seja aquele com o pokémon o atributo Speed mais alto está funcionando. A seguir segue-se os prints junto com a verificação do teste:

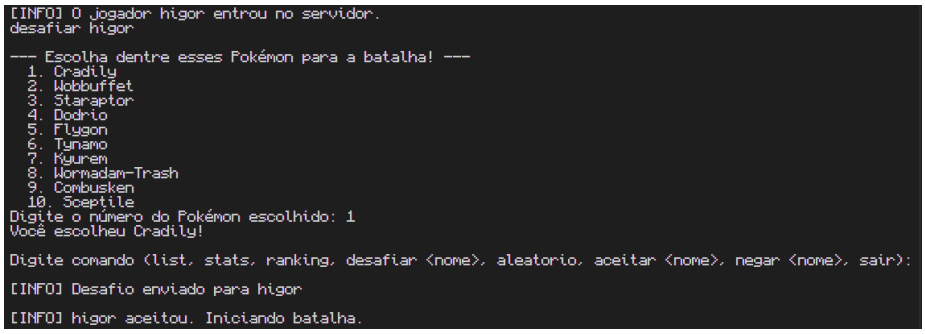

Figura 20: Terminal de Eduardo, mostrando se o golpe utilizado pelo seu pokémon foi efetivo ou não - Parte 1

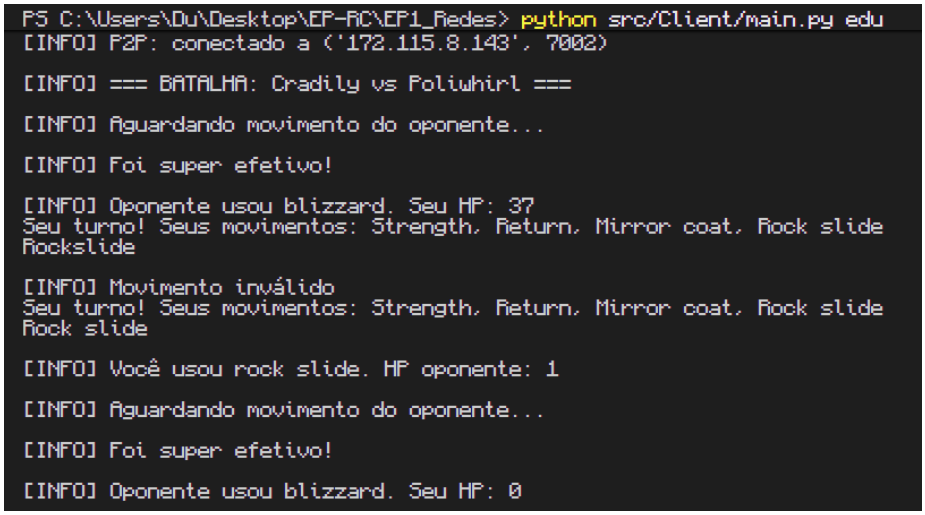

Figura 21: Terminal de Eduardo, mostrando se o golpe utilizado pelo seu pokémon foi efetivo ou não - Parte 2

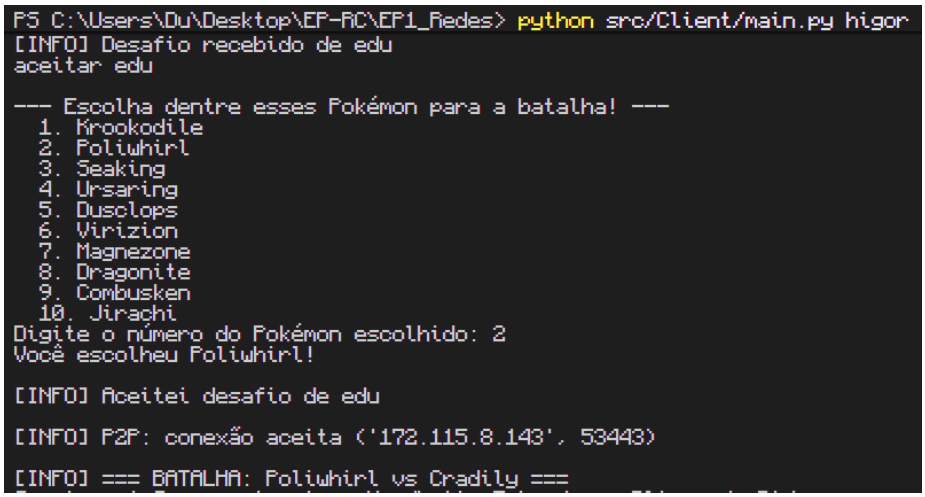

Figura 22: Terminal de Higor, mostrando se o golpe utilizado pelo seu pokémon foi efetivo ou não - Parte 1

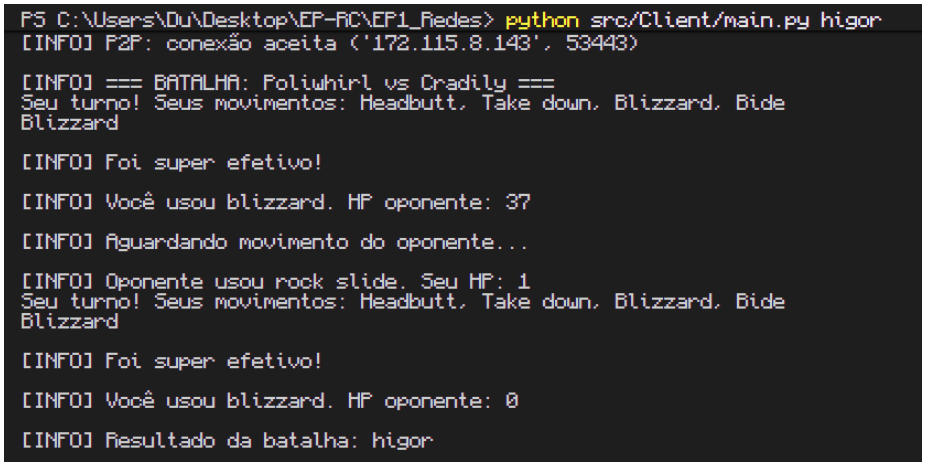

Figura 23: Terminal de Higor, mostrando se o golpe utilizado pelo seu pokémon foi efetivo ou não - Parte 2

Pois dos logs desses terminais, podemos fazer a verificação precisa se o sistema de tipos e fraquezas está funcionando, além de verificar a iniciativa dos jogadores com base na Speed de seus respectivos pokémons. Primeiramente, iremos checar o HP, Attack e Defense de Golem e Turtwig, além do ataque thunder.

Figura 24: Atributos do pokémon Poliwhirl

Figura 25: Atributos do pokémon Cradily

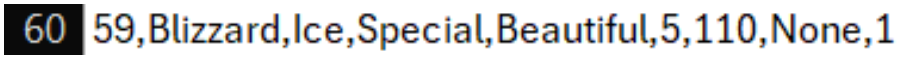

Figura 26: Atributos do ataque blizzard

Vemos que Poliwhirl possui um atributo de Speed maior que Cradily (90 43), logo, tudo correto em relação a iniciativa dos jogadores com base na Speed de seus pokémons, visto que o jogador com o pokémon com o atributo de Speed mais alto (higor) foi o primeiro a agir na batalha.
Vemos que Blizzard um ataque Special, logo devemos verificar o Special Attack do pokémon atacante o Special Defense do pokémon que sofreu o golpe. Vemos que o Poliwhirl possui 50 de Special Attack. Crudily por sua vez possui 86 de Hp e 107 de Special Defense . O ataque Blizzard utilizado por Poliwhirl, possui 110 de Power. Agora iremos aos cálculos! Em **battle.py**, utilizamos a seguinte fórmula para calcular o dano:

```
# em battle.py
 damage = (((2 * 50 / 5 + 2) * power * (attack / defense)) / 50 + 2) *
 stab * type_effectiveness
```

type_effectiveness é calculado por meio do método **type_multiplier** da classe Move em **move.py**. Como Blizzard é do tipo Ice, e Cradily é do tipo Grass, logo a type_effectiveness é de 2.
Em relação a stab, calculada em **battle.py**, ela será de 1.5 se o tipo do ataque estiver contido nos tipos do pokémon, e será apenas 1 caso contrário. Como Blizzard (Ice) não está contido nos tipos de Poliwhirl (Water), o stab será de 1.
Com essas informações podemos efetivar o cálculo para ver se o sistema de fraqueza e tipos está funcionando de fato:

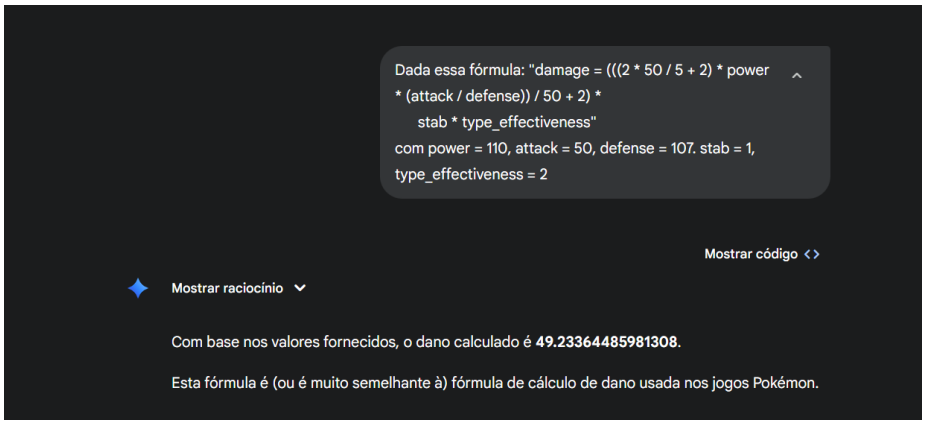

Figura 27: Cálculo da fórmula de ataque feito pela IA Gemini

Logo, como o dano é truncado para baixo, ele foi de 49. Subtraindo com o Hp de Cradily, temos 86 - 49 = 37. Mesmo valor de Hp mostrado no log do terminal, logo, nosso sistema de tipos e fraquezas está funcionando corretamente.

- **Teste de Convite de Desafio Expirado:** Na nossa implementação, se após um desafio for feito, e o jogador que receber o desafio não aceitar, ele expira.

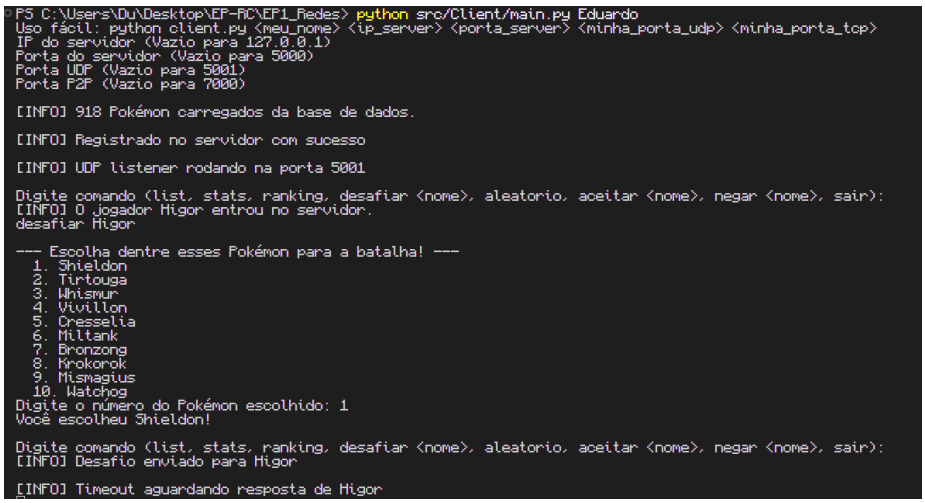

Figura 28: Terminal de Eduardo mostrando o timeout de seu convite para batalhar Higor

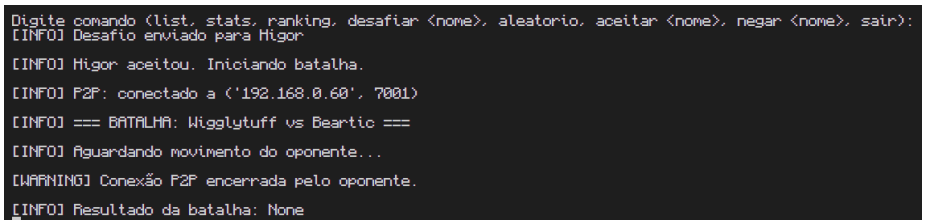

Figura 29: Terminal de Higor mostrando a tentativa de aceitar o convite de Desafio de Eduardo após o timeout ocorrer, o resultando na impossibilidade de entrar na batalha

Podemos ver que a expiração do convite está funcionando corretamente. Um ponto de melhoria seria a mensagem de timeout também aparecer para o jogador que foi desafiado.

- **Teste de Batalha Encerrada por Inatividade:** Também implementamos um sistema onde a batalha é encerrada caso o jogador que irá agir no turno atual fique muito tempo sem agir. Após cerca de 1 minuto, a batalha foi devidamente encerrada, confirmando que nosso sistema está funcionando.

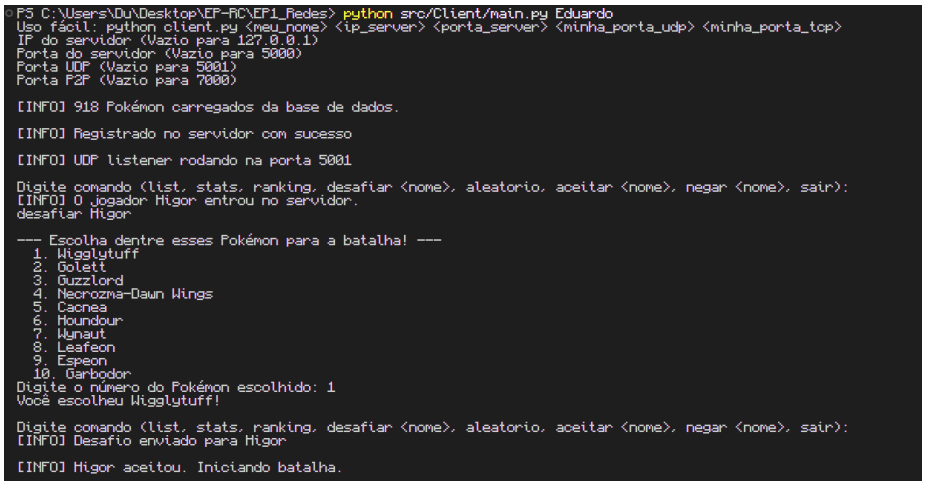

Figura 30: Terminal de Eduardo mostrando o encerramento da batalha por inatividade - Parte 1

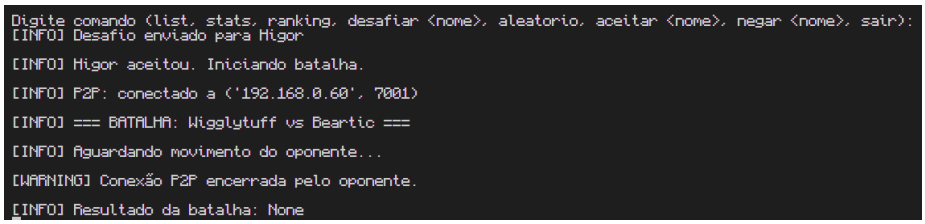

Figura 31: Terminal de Eduardo mostrando o encerramento da batalha por inatividade - Parte 2

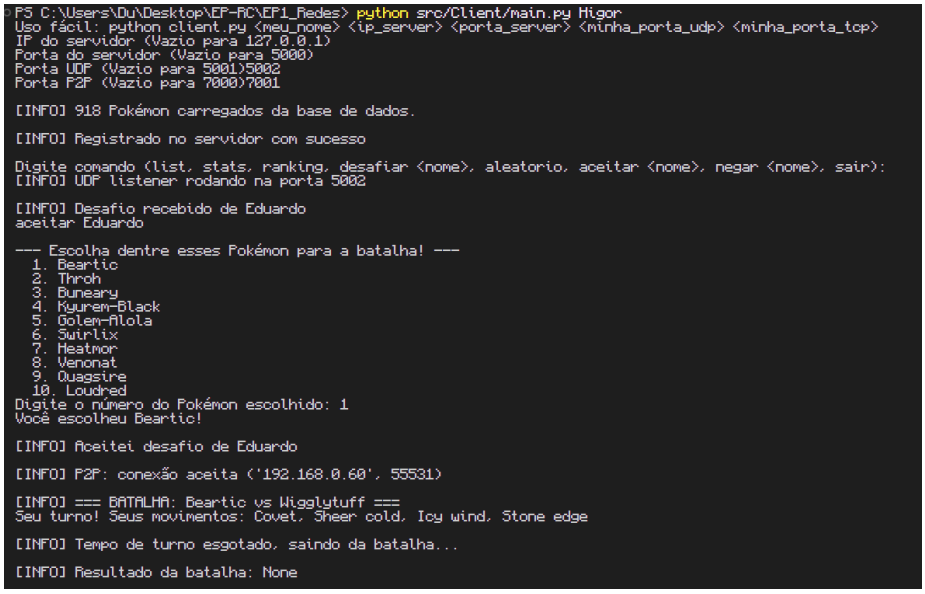

Figura 32: Terminal de Higor mostrando o encerramento da batalha por inatividade

- **Tratamento de duplicada:** Um cliente se conectou utilizando o nome A. Após isso, um segundo cliente tentou se conectar usando o mesmo nome. o servidor corretamente rejeitou a segunda conexão e retornou a mensagem de erro {`'type': 'ERR', 'msg': 'name_in_use'`} ao segundo cliente. O log do servidor confirmou o recebimento da segunda tentativa e a não efetivação do registro, provando que a validação está funcionando.
  OBS: Embora ele esteja mais abaixo, este foi o primeiro teste que realizamos, por isso que as estatísticas de Eduardo, Higor, e de outros jogadores que foram carregadas no exemplos anteriores não foram carregadas pelo servidor, pois eles ainda não haviam se registrado nem batalhado.

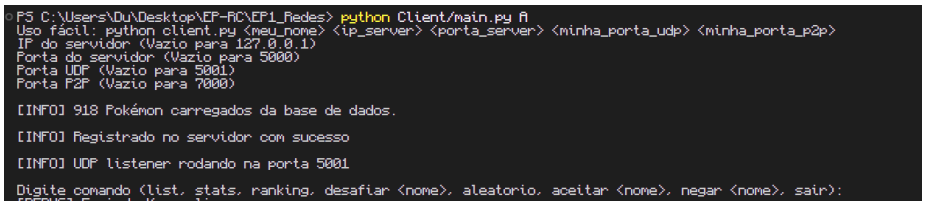

Figura 33: Jogador A original

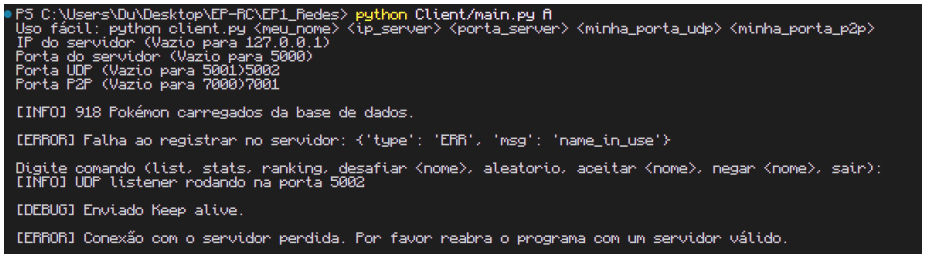

Figura 34: Jogador tentando conectar com um nome já em uso (A).

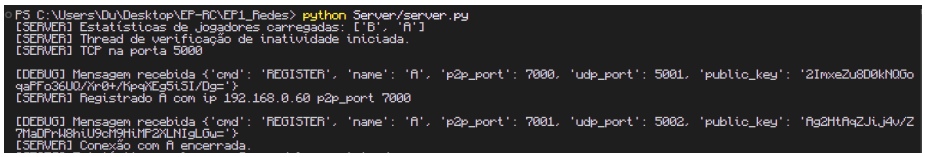

Figura 35: Log do servidor lidando com a duplicada

- **Teste de Conectividade via Internet:** Para validar o funcionamento da aplicação em um cenário real, foi realizado um teste de conexão através da internet. O objetivo era garantir que um cliente em uma rede externa conseguisse se conectar ao servidor localizado em outra rede, atrás de um roteador NAT.
  Para isso, Higor iniciou o servidor em sua máquina local e, para que a conexão externa fosse possível, ele configurou uma regra de Redirecionamento de Portas (Port Forwarding) em seu roteador. Essa regra direcionava todo o tráfego que chegasse na porta pública `25565` para a porta e IP correspondentes na máquina local onde o servidor estava rodando.
  Em seguida, Eduardo usando o nome Edj (erro de digitação, era para ser Edu), em uma rede completamente diferente, executou o cliente e, ao invés de usar um IP local como `127.0.0.1`, inseriu o endereço IP público da rede de Higor (`177.81.68.92`), especificando a porta `25565`.
  O teste foi um sucesso absoluto. O cliente de Eduardo conseguiu se conectar e se registrar no servidor de Higor, e o log do servidor confirmou

o recebimento da conexão a partir do IP público de Eduardo. Este teste comprova que a arquitetura cliente-servidor do projeto é robusta e funcional para ser utilizada através da internet.

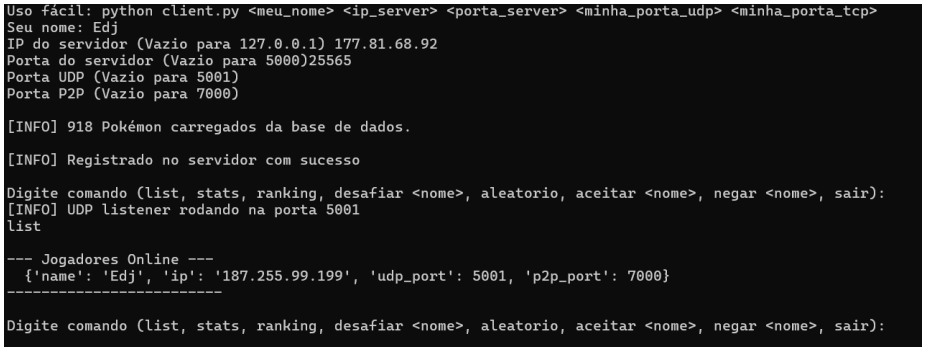

Figura 36: Cliente Edj executado por Eduardo

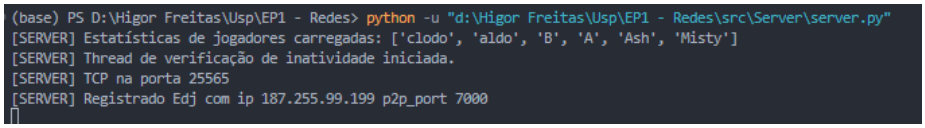

Figura 37: Servidor iniciado por Higor em sua máquina **Windows 10 Pro 22H2 (Build 19045.6456) — 64 bits**

- **Teste de Compatibilidade Multiplataforma:** Foi feito também um teste para validar se a aplicação funcionava corretamente em sistemas operacionais diferentes, confirmando a portabilidade do projeto.
  Para isso, o servidor foi executado em uma máquina, enquanto o Jogador A se conectou a partir de um computador com **Windows 10 Pro 22H2 (Build 19045.6456) — 64 bits** e o Jogador B a partir de um com **Arch Linux (Kernel 6.17.2-zen1-1-zen) - 64 bits**.
  Ambos os jogadores conseguiram se registrar no servidor, se desafiar e completar uma batalha P2P sem apresentar nenhum erro ou problema de compatibilidade. O teste foi um sucesso, e o log do servidor confirmou o registro e a conclusão da batalha entre os clientes nos diferentes sistemas.

Figura 38: Terminal do Jogador A, rodando em ambiente Windows 10.

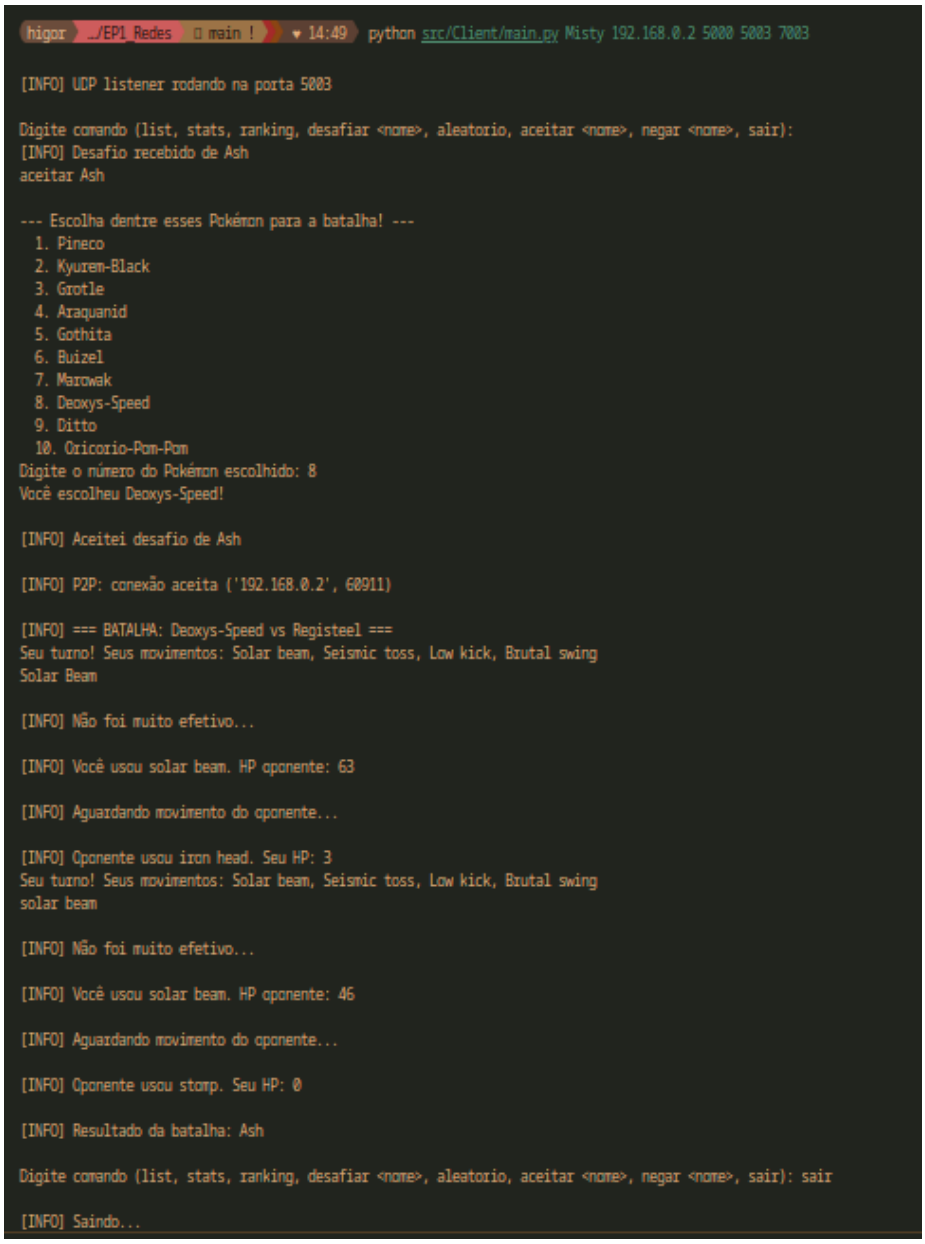

Figura 39: Terminal do Jogador B, rodando em Linux Arch.

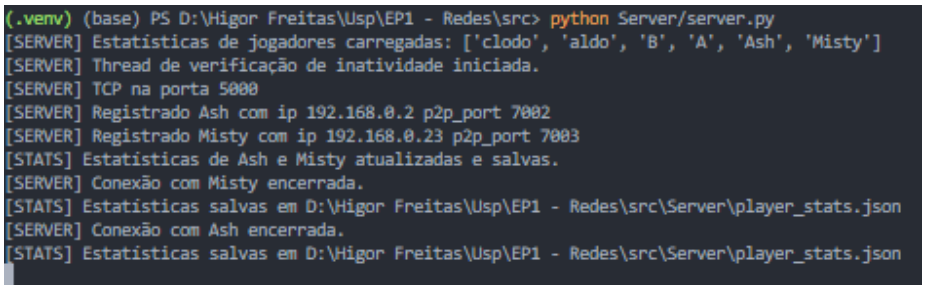

Figura 40: Log do servidor confirmando o registro e a batalha entre os clientes em sistemas operacionais diferentes.

7 Funcionalidades não implementadas e bugs

Por restrições de tempo e para focar nos objetivos de rede do projeto, algumas funcionalidades planejadas não foram implementadas, além de termos percebido uma possível vulnerabilidade com o nosso sistema:

7.1 Funcionalidades Planejadas

- **Contatos:** Uma funcionalidade planejada era um sistema de "contatos", onde um cliente poderia salvar a chave pública e o endereço de um amigo para iniciar uma batalha amistosa sem depender do servidor. Isso daria um propósito adicional à arquitetura de chave pública/privada.
- **Validação de Vitória no Servidor:** Para previnir trapaças, pretendíamos criar um sistema de hash cumulativo do estado da batalha. O vencedor enviaria a sequência de hashes para o servidor, que poderia validá-la para confirmar que o resultado é legítimo, sem precisar processar a batalha inteira.
- **Chat na Batalha:** Adicionar um canal de comunicação de texto simples dentro da conexão P2P criptografada.
- **Acurácia e Efeitos de Status:** Embora os movimentos possuam um atributo de acurácia (*accuracy*) carregado do nosso banco de dados, a lógica para que um ataque pudesse errar não foi implementada. Da mesma forma, movimentos que aplicam condições especiais (como *poison* ou *paralysis*) foram simplificados para causar apenas dano direto. Tais implementações deixariam o jogo mais divertido e interessante, porém não tivemos tempo de acrescenta-las.

7.2 Bugs e Vulnerabilidades Conhecidas

- **Vulnerabilidade de Confiança no Cliente:** O principal erro de design do sistema atual é que o servidor confia totalmente nas informações

enviadas pelos clientes. Como toda a lógica da batalha e o reporte do resultado são gerenciados pelo cliente, um usuário com um cliente modificado poderia facilmente manipular o jogo. Seria possível, por exemplo, alterar o dano dos seus golpes, ignorar o dano recebido ou simplesmente enviar uma mensagem de vitória para o servidor sem nunca ter batalhado. A implementação do hash cumulativo seria a solução para essa vulnerabilidade.

8 Diálogos com IA

Todos os diálogos feitos pelo membro do grupo Eduardo Almeida com a IA Gemini, tanto em relação a dúvidas no código, cálculo da fórmula de dano para o print, e em relação a dúvidas e problemas de erro com o LaTex foram documentadas em um arquivo PDF chamado **DialogosComIA** em anexo junto com este Exercício Prático.

Referências

- Documentação oficial da linguagem Python 3 para consulta de módulos como `socket`, `threading` e `json`.
- Documentação da biblioteca `cryptography` para Python, utilizada para a troca de chaves X25519 e a criptografia AES-GCM. Disponível em: https://cryptography.io/
- Fóruns de discussão como Stack Overflow para solução de problemas pontuais relacionados a sockets, threads e tratamento de exceções de rede.
- Repositório onde está o código do projeto: `https://github.com/HigorFr/EP1_Redes`.
- IA Gemini para retirar dúvidas em relação ao código, ajudar a implementar o sistema de registro de vitórias e derrotas; auxiliar na correção do problema de leitura de teclado que impedia que as batalhas fossem iniciadas; auxiliar no entendimento da causa e correção da dessincronização do Hp durante a batalha; auxiliar na implementação do ranking geral; realizar o cálculo da fórmula de dano para o print no Teste de Fraquezas e ajuda no geral com a formatação e erros do LaTeX durante a elaboração deste relatório e do PDF **DialogosComIA**.