INSTITUTO FEDERAL DE EDUCAÇÃO, TECNOLOGIA E CIÊNCIA DO RIO GRANDE DO NORTE

Docente: Jefferson Igor Duarte Silva

Turma: Informática Vespertino – 2° ano (INFO2V)

Discentes: Sofia Guimarães (20241054010011), Bárbara Cavalcante (20241054010003), Daniel Guimarães (20241054010076), Ybison de Oliveira (20241054010023), Fernanda da Silva (202410540061)

RELATÓRIO DE REDES SOBRE A IMPLEMENTAÇÃO DE APLICAÇÃO P2P COM SELEÇÃO DE PROTOCOLO DE TRANSPORTE E SUPORTE A IPV4/IPV6

IPANGUAÇU/RN – 2025.1

INTRODUÇÃO

- Arquitetura da Solução:

O projeto implementa um jogo da forca multiplayer em rede usando arquitetura P2P (peer-to-peer), onde um jogador atua como servidor e outro como cliente, podendo se conectar via TCP ou UDP.
O sistema é dividido em três módulos principais, os arquivos *.py* — cada um com responsabilidades bem definidas — e uma camada de interação com a rede.

1. main.py

- **Função principal:** Orquestrar o sistema.
- **Camada de Interface (main.py):** provê uma interface **Tkinter** para o usuário. Nela o jogador pode configurar se será **servidor** ou **cliente**, escolher **TCP ou UDP**, inserir endereço IP/porta, e durante a partida visualizar:
  - o boneco,
  - o estado da palavra,
  - letras erradas,
  - tentativas restantes,
  - e usar um pequeno **chat P2P** embutido.
- **Responsabilidades:**
  - Inicia a interface gráfica (*ForcaGUI*).
  - Coleta informações de configuração (modo servidor/cliente, IP, porta, protocolo) via interface Tkinter.
  - Chama funções de inicialização do jogo e da rede.
  - Faz o "link" entre a lógica de jogo (*jogo.py*) e a camada de comunicação (*p2p.py*).
- **Interação:**
  - Depende de *ForcaGUI* para renderizar a interface.
  - Usa classes de *p2p.py* para enviar/receber dados.
  - Usa a classe *Forca* de *jogo.py* para gerenciar as regras do jogo.
- **Fluxo:** funciona da maneira descrita no passo-a-passo abaixo.
  - O servidor inicia, define a palavra e envia o estado inicial ao cliente.
  - O cliente se conecta, recebe o estado e passa a enviar suas jogadas.
  - O servidor processa cada jogada, atualiza o jogo (*Forca*) e envia de volta o estado atualizado.
  - O cliente reflete essas mudanças na interface.

2. ForcaGUI (Tkinter) — Interface Gráfica

- **Função principal:** Intermediar a comunicação entre **usuário**, **jogo** e **rede**.
- **Responsabilidades:**
    - Captura eventos do usuário (entrada de letras, mensagens de chat, botões).
    - Atualiza elementos visuais do jogo (palavra, tentativas, boneco, chat).
    - Controla as telas iniciais e de vitória/derrota.
    - Encaminha jogadas para o outro jogador via rede.
- **Interação:**
    - **Com o usuário:** Recebe comandos e envia feedback visual.
    - **Com *jogo.py*:** Invoca métodos para validar tentativas e atualizar estado.
    - **Com *p2p.py*:** Envia mensagens de jogo e chat pela rede.
    - **Com *main.py*:** Recebe parâmetros iniciais e dispara a lógica do servidor ou cliente.

3. jogo.py (Forca) — Lógica do Jogo

- **Função principal:** Implementar as regras da forca.
- **Responsabilidades:**
    - Armazenar palavra secreta, dica, letras acertadas e erradas.
    - Controlar número de tentativas restantes.
    - Determinar vitória, derrota e progresso da palavra.
    - Fornecer representação visual do boneco ASCII conforme erros.
- **Interação:**
    - É instanciado pela interface (*ForcaGUI*).
    - Recebe tentativas do jogador pela GUI.
    - Retorna estado atualizado do jogo para exibição e envio pela rede.

4. p2p.py (P2PServer / P2PClient) — Comunicação em Rede

- **Função principal:** Garantir a troca de dados entre jogadores.
- **Responsabilidades:**
    - Criar conexões TCP ou UDP.
    - Enviar mensagens de controle, jogadas e chat.

    - Receber mensagens e repassar para a interface.
    - Suportar IPv4 e IPv6.
- **Interação:**
    - **Com a rede:** Lida diretamente com sockets para transmissão.
    - **Com ForcaGUI:** Entrega mensagens recebidas e envia mensagens produzidas pela interface.
    - **Com main.py:** É configurado com IP, porta e protocolo escolhidos.

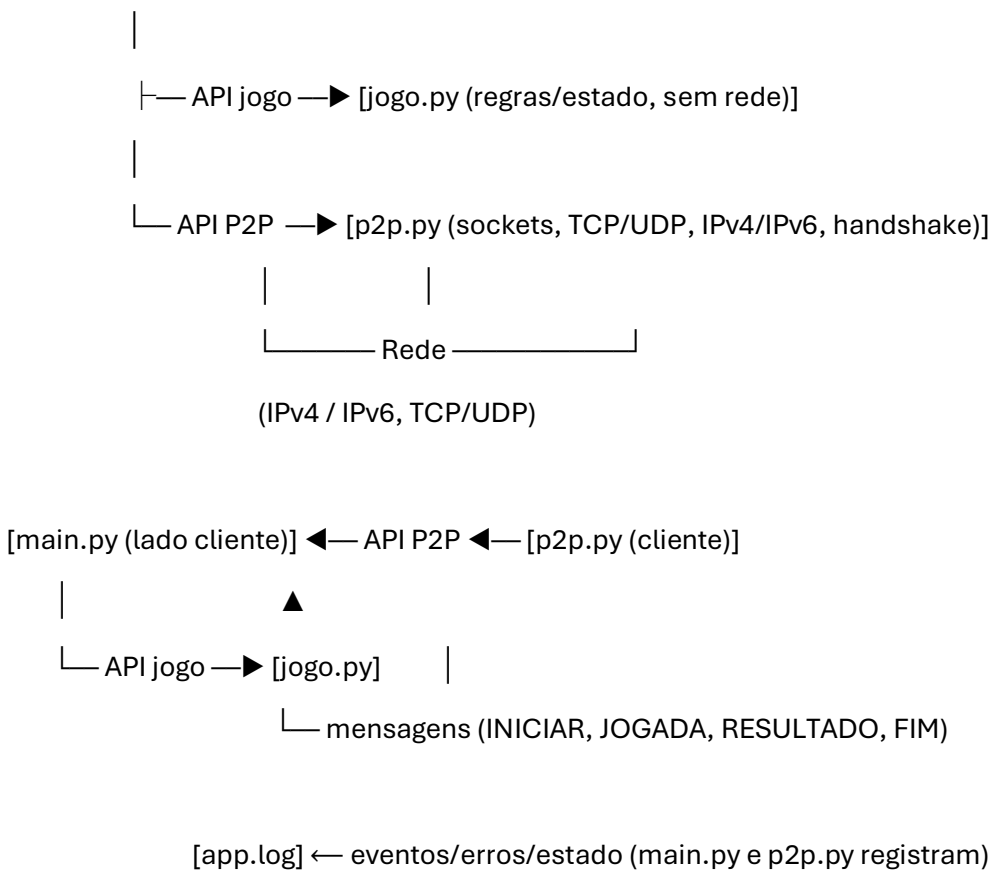
5. Rede (TCP/UDP)

- **Função principal:** Meio físico/lógico para a troca de mensagens.
- **Características:**
    - **TCP:** Conexão orientada, garante ordem e integridade dos pacotes.
    - **UDP:** Conexão não orientada, mais rápida, mas sem garantias de entrega.
- **Interação:**
    - Recebe dados do *p2p.py* e entrega ao outro jogador.
    - Retorna dados recebidos para o *p2p.py*.

Fluxo de funcionamento

1. **Configuração**: Usuário escolhe se será servidor ou cliente, define IP, porta e protocolo.
2. **Conexão**: *p2p.py* estabelece o canal de comunicação P2P.
3. **Inicialização do jogo**:
    a. Servidor define a palavra e dica.
    b. Cliente recebe o estado inicial via rede.
4. **Gameplay**:
    a. Jogador envia tentativa – *ForcaGUI* chama *jogo.py* (no servidor) ou envia para o servidor via *p2p.py* (no cliente).
    b. Estado atualizado é enviado de volta para o outro jogador.
5. **Chat**: Mensagens de texto seguem o mesmo caminho da jogada, mas não alteram o estado do jogo.
6. **Fim de jogo**: *jogo.py* detecta vitória ou derrota, *ForcaGUI* exibe resultado.

[iniciar.bat] —▶ [main.py (UI/CLI)]



DESENVOLVIMENTO:

- ## Desafios Encontrados:

Os principais desafios que enfrentamos no desenvolvimento do projeto estavam relacionados à interface gráfica e a implementação do protocolo IPv6. Todos esses estavam diretamente relacionados a problemas nos códigos dos arquivos previamente mencionados.

1. Implementação do IPv6:

Quanto a implementação do IPv6, estamos tendo problemas para associar cliente e servidor ao protocolo. Acontece que, para integrar o IPv6, devemos organizar em camadas bem separadas, cada arquivo com uma função clara. O *main.py* funciona como ponto de entrada e interface de linha de comando. É nele que o usuário escolhe se quer ser host ou cliente, qual protocolo de transporte vai

usar (TCP ou UDP) e em qual endereço e porta vai se conectar. Esse arquivo não se preocupa com detalhes de rede: ele apenas coleta parâmetros do usuário, chama funções de rede do módulo *p2p.py* e depois entrega as jogadas para a lógica do jogo, que está isolada em *jogo.py*.

O arquivo *p2p.py* concentra toda a parte de rede e de arquitetura P2P. Ele precisa resolver se a comunicação será feita em IPv4 ou IPv6, mas o restante da aplicação não precisa saber disso. Para isso, o *p2p.py* usa a função *socket.getaddrinfo*, que já retorna automaticamente a família correta (AF_INET para IPv4 e AF_INET6 para IPv6). Assim, não é necessário o usuário distinguir manualmente — basta passar o endereço e a porta. Ou seja, todos os arquivos, com exceção do *jogo.py*, que é um arquivo totalmente independente da rede, estão conectados ao IPv6.

Demorou muito até descobrirmos o que de fato havia sido o problema. A cada checagem, não parecia haver nada de errado, até lembrarmos que não foi feita a ligação com o arquivo *.bat*. O arquivo em lote *iniciar.bat* serve para facilitar a execução. Ele pode encapsular comandos já prontos para diferentes cenários: hospedar uma partida TCP/IPv4, hospedar uma partida TCP/IPv6, conectar-se como cliente em IPv6, e assim por diante. Esse script também redireciona a saída do programa para um arquivo *.log*, de forma que todo o histórico de execução fique registrado automaticamente. É importante enfatizar que o *.bat* não cria ou trata a conexão em si; ele apenas repassa a string do endereço para o Python, isso é feito usando colchetes ([]) e dois pontos (:). O uso de colchetes é o que garante que endereços IPv6 sejam aceitos corretamente. O que fizemos foi definir essa string de endereço (::), que seria passada para o servidor, e depois fosse requisitado o endereço no cliente do próprio IPv6, em hexadecimal.

2. Problemas na interface gráfica:

Não sabíamos como implementar a interface gráfica, parte muito importante do projeto em si. Sem ela, o jogo inteiro era rodado no terminal do console – mesmo lugar onde digita-se informações de conexão, como os protocolos, endereço de IP etc.

Para resolver o problema, fizemos várias pesquisas sobre a integração da interface gráfica ao projeto. Para implementá-la, salvamos o *main.py* com a implementação da GUI com o Tkinter. Isso fez com que, ao iniciar como cliente, em vez de aparecer o texto do console no terminal pedindo letras, uma janela gráfica surgisse com: o boneco da forca, a palavra mascarada letras erradas, tentativas restantes, um campo digital para digitar letras e enviar.

Outro problema que enfrentamos sobre a interface gráfica foi com relação ao boneco que aparece no jogo. Este não era exibido ao abrir a interface do jogo, bem como a palavra e o contador de tentativa. O problema era que o servidor já inicializava com espaços para digitar as respectivas informações oriundas do próprio jogo, mas que, entretanto, não podiam ser inseridas porque o cliente também precisava ser conectado. O boneco, a palavra que estava sendo formada, bem como o contador de tentativas eram todos inerentes à parte do cliente inacessível sem o *login*.

Para resolver o problema, ajustamos o *main.py* para que, no modo servidor, primeiro aparecesse pedindo as informações de autenticação, a palavra secreta e a dica. Com isso, o servidor cria um objeto forca no programa e esse retorna "INICIAR JOGO" para o cliente. A interface principal aparece, então, atualizada com o boneco, a palavra e as tentativas.

Um terceiro problema que enfrentamos ainda na interface gráfica foi relacionado à dica do jogo – esta não estava sendo exibida. O motivo dela não aparecer era que na interface do Tkinter não havia sido criado um *label* para exibi-la na tela – estava apenas sendo enviada pela rede e armazenada no objeto *jogo* e não era visível na tela sob estas condições.

A solução para o problema era bastante simples. Apenas era preciso criar um *label* que contivesse a dica logo abaixo da palavra e atualizar esse mesmo *label* à medida que a tela fosse sendo atualizada, ao longo das tentativas. Para isso, fizemos algumas alterações no código, especificamente na função *atualizar_tela*.

CONCLUSÃO:

- **Análise Comparativa (TCP e UDP):**

1. Diferenças na Implementação

TCP (Transmission Control Protocol):

- **Conexão**: Estabelece uma conexão persistente entre cliente e servidor

- **Handshake**: Utiliza o handshake padrão do TCP (3-way handshake)
- **Fluxo de dados**: Bidirecional através da mesma conexão
- **Controle de sessão**: A conexão permanece ativa durante toda a partida

Python:

```
# Servidor TCP
self.sock.listen(1)
conn, addr = self.sock.accept()  # Aceita conexão
self.sock = conn                 # Substitui socket pelo da
conexão

# Cliente TCP
self.sock.connect((self.ip, self.porta))  # Conecta diretamente
```

UDP (User Datagram Protocol):

- **Conexão**: Sem estabelecimento formal de conexão
- **Handshake**: Implementa um handshake customizado (HELLO|)
- **Fluxo de dados**: Baseado em datagramas independentes
- **Controle de sessão**: Mantém endereço do cliente para envio de respostas

Python:

```
# Servidor UDP
# Aguarda HELLO do cliente para descobrir seu endereço
while True:
    msg = srv.recv()
    if msg and msg.startswith("HELLO|"):
        break

# Cliente UDP
cli.send("HELLO|")  # Envia HELLO para se identificar
```

2. Análise do Comportamento com Perda de Pacotes UDP

Problema Crítico: Ausência de Confiabilidade

Quando um pacote UDP contendo uma jogada é perdido, a aplicação não possui mecanismo de recuperação. Isso pode causar diversos problemas:

Cenário 1: Perda de Jogada do Cliente

```
Cliente envia: "JOGADA|A"
[PACOTE PERDIDO]
Servidor: Nunca recebe a jogada
Cliente: Fica aguardando resposta indefinidamente
```

Consequência: O jogo trava, pois o cliente nunca recebe o estado atualizado.

Cenário 2: Perda de Estado do Servidor

```
Cliente envia: "JOGADA|A"
Servidor processa e envia: "STATE|A___|E,R,R|4|dica|CONT"
[PACOTE PERDIDO]
Cliente: Nunca recebe o novo estado
```

Consequência: Cliente e servidor ficam dessincronizados.

Cenário 3: Perda de Mensagem de Chat

```
Cliente envia: "CHAT|Olá!"
[PACOTE PERDIDO]
Servidor: Nunca exibe a mensagem
```

Consequência: Comunicação incompleta entre os jogadores.

Como a Aplicação Lida com Perdas:

A aplicação atual **não implementa nenhum mecanismo de recuperação**:

1. **Sem *acknowledgments***: Não há confirmação de recebimento
2. **Sem retransmissão**: Pacotes perdidos nunca são reenviados
3. **Sem *timeout* de aplicação**: Não detecta quando uma resposta não chega
4. **Sem numeração de sequência**: Impossível detectar pacotes perdidos ou duplicados

3. Comparação Detalhada

Aspecto	TCP	UDP
Confiabilidade	✅ Garantida pelo protocolo	❌ Não garantida
Ordem dos dados	✅ Preservada	❌ Não garantida
Controle de fluxo	✅ Automático	❌ Inexistente
Detecção de erros	✅ Automática com retransmissão	❌ Básica, sem recuperação
Overhead	⚠️ Maior (headers + controle)	✅ Menor
Complexidade	✅ Simples para o desenvolvedor	❌ Complexa (requer implementação manual)
Latência	⚠️ Maior devido ao controle	✅ Menor
Adequação para jogos	✅ Ideal para jogos por turnos	❌ Inadequado sem implementações extras

4. Problemas Específicos Identificados no Código

Timeout Inadequado

Python:

```
# Ambos os protocolos usam o mesmo timeout
self.sock.settimeout(self.timeout)  # 60s padrão
```

Para UDP, este timeout é insuficiente para detectar pacotes perdidos rapidamente.

Falta de Validação de Estado

O código não verifica se o estado recebido é consistente com o estado anterior, permitindo dessincronização.

Handshake UDP Frágil

Python:

```
if self.protocolo == UDP:
    cli.send("HELLO|")
```

Se o HELLO| for perdido, a conexão nunca é estabelecida.

5. Protocolo Mais Adequado: TCP

Por que TCP é superior para esta aplicação:

1. Natureza do Jogo

- Jogo por turnos (não requer baixa latência)
- Estado crítico que não pode ser perdido
- Sequência de jogadas importante

2. Requisitos de Confiabilidade

- Cada jogada deve ser processada exatamente uma vez
- Estado do jogo deve permanecer sincronizado
- Mensagens de chat não podem ser perdidas

3. Simplicidade de Implementação

- TCP elimina a necessidade de implementar:
    - Controle de retransmissão
    - Detecção de duplicatas
    - Ordenação de mensagens
    - Controle de fluxo

4. Robustez

- Detecta automaticamente desconexões
- Recupera-se de problemas de rede
- Garante integridade dos dados

Evidências do Log

```
[2025-08-15 06:25:37] ERRO_SERVIDOR: timed out
```

O log mostra timeouts, sugerindo problemas de conectividade que são melhor tratados pelo TCP.

Solução ideal: Usar TCP exclusivamente para esta aplicação, eliminando a complexidade desnecessária e garantindo uma experiência de usuário confiável.

O TCP se mostra significativamente mais adequado para o jogo da Forca P2P devido à natureza crítica do estado do jogo e à necessidade de comunicação confiável. A implementação atual com UDP possui falhas fundamentais que podem resultar em travamentos e dessincronização, tornando a experiência do usuário frustrante e não confiável.