UNIVERSIDADE FEDERAL DE ALAGOAS - UFAL
CIÊNCIA DA COMPUTAÇÃO
REDES DE COMPUTADORES

JOGO DE ADIVINHAÇÃO

Érico Almeida Bernardino,
Guilherme Alessander da Silva Fernandes
João Victor Duarte do Nascimento,
Rian Antonio da Silva Gaião

Maceió, Novembro de 2025

RESUMO

Este documento apresenta os detalhes e a arquitetura de um jogo multiplayer de adivinhação desenvolvido em Python, utilizando as bibliotecas **`Socket`** e **`Threading`**. O objetivo é demonstrar a comunicação TCP entre um servidor central e múltiplos clientes, aplicada em um ambiente real. O servidor gerencia o estado do jogo e utiliza um sistema de ***broadcast*** para notificar todos os participantes (conforme **`servidor.py`**). Os clientes podem se conectar simultaneamente, enviar tentativas e receber feedback em tempo real (conforme **`cliente.py`**). A comunicação é padronizada através de um protocolo personalizado (definido em **`protocolo.py`**).

Alunos:

Érico Almeida,
Guilherme Alessander da Silva Fernandes
João Victor Duarte do Nascimento,
Rian Antonio da Silva Gaião

Orientador:

Prof. Almir Pereira Guimarães
Universidade Federal de Alagoas – UFAL

Visto do Orientador: _____

SUMÁRIO

Introdução

Este projeto consiste em um jogo multiplayer de adivinhação onde múltiplos clientes competem para adivinhar um número secreto escolhido pelo servidor (conforme `README.md`).

A aplicação é construída sobre o protocolo TCP, que especifica um fluxo de dados confiável e orientado, identificado no código como `SOCK_STREAM`. O servidor utiliza `AF_INET` para indicar o uso do protocolo IPv4.

Para permitir que múltiplos clientes joguem simultaneamente, o servidor utiliza *threads*. Uma *thread* é um fluxo de execução que permite que a aplicação execute múltiplas tarefas concorrentemente. No `servidor.py`, o servidor aceita uma nova conexão (`.accept()`) e inicia uma nova `threading.Thread` (conforme `servidor.py`) para gerenciar a comunicação com aquele cliente específico, enquanto a *thread* principal continua aguardando novas conexões (`.listen()`).

2. ARQUITETURA DO CÓDIGO

O projeto é dividido em três arquivos principais:

2.1. Protocolo.py

Este arquivo define a classe `Protocolo`, que padroniza todas as mensagens trocadas entre o servidor e os clientes. Isso garante que ambos os lados saibam como interpretar os dados recebidos.

- **Métodos Estáticos:** A classe utiliza `@staticmethod`, um "decorador" que define métodos que não recebem uma referência de instância (`self`).
- `codificar(comando, dados="")`: Recebe um comando (ex: "TENTATIVA") e dados (ex: "50") e os formata em uma única string, usando "|" como separador: `"TENTATIVA|50"` (conforme `protocolo.py`).
- `decodificar(mensagem)`: Recebe a string formatada (ex: `"MAIOR|O número é menor"`) e a divide (`.split("|", 1)`) para retornar o comando e os dados separadamente (conforme `protocolo.py`).
- **Comandos Definidos:**
    - Cliente -> Servidor: `TENTATIVA`, `SAIR` (conforme `protocolo.py`).
    - Servidor -> Cliente: `INICIAR`, `MAIOR`, `MENOR`, `ACERTOU`, `FIM_PARTIDA`, `ERRO` (conforme `protocolo.py`).

2.2. Servidor.py

O `servidor.py` é o núcleo do jogo. Ele gerencia o estado da partida e a comunicação com todos os clientes conectados.

- **Classe Jogo:**
  - `__init__`: Inicializa as variáveis do jogo, incluindo a lista de `clientes` conectados (conforme `servidor.py`).
  - `inicar_game()`: Define o `jogo_ativo` como `True` e sorteia um novo `num_secreto` entre 1 e 100 (conforme `servidor.py`).
  - `broadcast(msg, cliente_origem=None)`: Envia uma mensagem (`msg`) para todos os clientes na lista `self.clientes`, exceto o cliente de origem (para evitar que o jogador receba a própria notificação de acerto, por exemplo). O método também remove clientes que podem ter se desconectado inesperadamente (conforme `servidor.py`).
- **Função `clientes(conexao, endereco)`:**
  - Esta função é executada em uma *thread* separada para cada cliente (conforme `servidor.py`).
  - Ela entra em um loop `while True` para receber dados (`.recv()`) e decodificar (`.decode()`) as mensagens daquele cliente.
  - **Lógica do Jogo:**
    - `TENTATIVA`: Converte os `dados` para inteiro. Compara a tentativa com o `jogo.num_secreto` e envia de volta `MAIOR`, `MENOR` ou `ACERTOU` (conforme `servidor.py`).
    - `ACERTOU`: Envia a mensagem `ACERTOU` ao jogador vencedor. Em seguida, usa `jogo.broadcast()` para notificar todos os outros jogadores sobre o fim da partida. Por fim, inicia um cronômetro de 5 segundos (notificando a contagem via *broadcast*) e chama `jogo.inicar_game()` para reiniciar a partida (conforme `servidor.py`).
    - `SAIR`: Envia uma mensagem de `FIM_PARTIDA` ao cliente e encerra o loop, o que levará ao fechamento da conexão (conforme `servidor.py`).

- **Função `main()`:**
    - Cria o socket TCP (`socket.socket(socket.AF_INET, socket.SOCK_STREAM)`) (conforme `servidor.py`).
    - Associa o servidor ao endereço `localhost` e à porta `8888` (`servidor.bind()`).
    - Começa a escutar por conexões (`servidor.listen()`).
    - Inicia o primeiro jogo (`jogo.inicar_game()`) (conforme `servidor.py`).
    - Entra no loop principal, aceitando novas conexões (`servidor.accept()`) e iniciando uma nova `threading.Thread` (direcionada à função `clientes`) para cada conexão (conforme `servidor.py`).

2.3. Cliente.py

O `cliente.py` permite que o usuário se conecte ao servidor e participe do jogo. Ele precisa gerenciar duas tarefas simultaneamente: ouvir o usuário (input) e ouvir o servidor (respostas).

- **Função `ouvir_server(cliente)`:**
  - Esta função é executada em uma *thread* separada (conforme `cliente.py`).
  - Ela fica em um loop `while True`, aguardando mensagens do servidor (`cliente.recv(1024).decode()`).
  - Quando uma mensagem chega, ela é decodificada (`Protocolo.decodificar()`) e exibida no console do usuário com (conforme `cliente.py`).
  - A *thread* é definida como `daemon = True`, o que significa que ela é uma tarefa de segundo plano e será encerrada automaticamente quando o programa principal (a *thread* de input) terminar.
- **Fluxo Principal (Main Thread):**
  - Cria o socket e se conecta ao servidor (`cliente.connect(('localhost', 8888))`) (conforme `cliente.py`).
  - Inicia a *thread* `ouvir_server` (conforme `cliente.py`).
  - Entra em um loop `while True` que captura o `input()` do usuário (conforme `cliente.py`).
  - Se o usuário digitar "sair", o comando `SAIR` é codificado e enviado (`.send()`), e o loop é interrompido (`break`) (conforme `cliente.py`).
  - Para qualquer outro texto, o programa envia o comando `TENTATIVA` com o número digitado (conforme `cliente.py`).
  - Pequenas pausas (`time.sleep(0.2)`) são usadas para melhorar a visualização e evitar a sobreposição de mensagens no console.

3. Ressalvas

3.1. O que poderia ter sido implementado

1. Robustez e Tratamento de Erros

A melhoria mais crítica seria tornar o servidor à prova de falhas causadas por entradas inesperadas do cliente.

- Validação de Entradas no Servidor:
- Validação de Entradas no Cliente

2. Protocolo de Comunicação

O protocolo atual (`comando|dados`) é simples, mas frágil

- Uma abordagem muito mais robusta e extensível é usar JSON.
- Realizar também tratamento de Buffers (Message Framing)

3. Arquitetura e Escalabilidade

O modelo de "uma thread por cliente" (`threading.Thread(target=clientes, ...)`) é ótimo, mas não é escalável.

- Usar `asyncio` (Programação Assíncrona):
- Múltiplas "Salas" de Jogo:

3.2 Principais Dificuldades no Projeto

Biblioteca Socket

A principal dificuldade foi compreender o fluxo de comunicação entre servidor e cliente. Como o protocolo TCP é orientado a conexão é baseado em fluxo contínuo de bytes, nem sempre as mensagens chegavam completas, exigindo o uso de métodos de codificação e decodificação padronizados. Além disso, erros como desconexões inesperadas e travamentos durante o envio ou recebimento de dados precisaram ser tratados para evitar que o servidor encerrasse de forma abrupta.

Biblioteca Threading

A implementação de múltiplas threads trouxe desafios relacionados à concorrência e sincronização. Cada cliente executa em uma thread independente, o que pode gerar condições de corrida caso várias threads tentem acessar ou modificar recursos compartilhados (como a lista de jogadores conectados) ao mesmo tempo. Também foi necessário compreender o comportamento das threads daemon, garantindo que o encerramento do programa ocorresse de forma controlada e sem bloqueios. Essas dificuldades ajudaram a consolidar o entendimento sobre comunicação em rede e execução paralela em Python, conceitos fundamentais para aplicações distribuídas.

REFERÊNCIAS

- **Threads em Python**, Lucas. *Criando e Gerenciando Threads em Python*. Medium, 2020. Disponível em: https://medium.com/@habbema/threads-em-python-9a3a7b3c776d. Acesso em: 28 out. 2025.
- **Um guia completo para a programação de sockets em Python**. *Serhii Orlivskyi*: https://www.datacamp.com/pt/tutorial/a-complete-guide-to-socket-programming-in-python. Acesso em: 28 out. 2025.