UNIVERSIDADE DE SÃO PAULO

ESCOLA DE ARTES, CIÊNCIAS E HUMANIDADES (EACH)

ACH2026 – Redes de Computadores

Exercício Programa - 2º semestre de 2025

Docente: Prof. Dr. Renan Cerqueira Afonso Alves

14671388	Guilherme Oliveira de Souza	guilherme27souza@usp.br
15574558	Enzo Zanetti Camargo	enzopcz@usp.br

São Paulo

2025

Exercício Programa - RC

O projeto atual se trata de um exercício programa da disciplina de Redes de Computadores do curso de Sistemas de Informação da Universidade de São Paulo - EACH, cujo objetivo era desenvolver uma aplicação de jogo da forca, criando o código para cliente e servidor.

Os arquivos da aplicação estão disponíveis no repositório no GitHub de link https://github.com/gui27souza/RC-EP/tree/main

O projeto foi desenvolvido pelos alunos Guilherme Oliveira de Souza e Enzo Zanetti Camargo.

Como executar?

Devido ao uso de módulos relativos em Python, é necessário seguir os seguintes passos para poder executar o programa:

Na raíz do diretório RC-EP/ (um nível acima de app/) execute o respectivo comando:

Para servidor:
python3 -m app.app_server.hangman-server <numero-de-jogadores> [<porta>]

Exemplos:
python3 -m app.app_server.hangman-server 10
python3 -m app.app_server.hangman-server 3 2727

Para cliente:
python3 -m app.app_client.hangman-client <nome-do-jogador> [<IP>:<Porta>]

Exemplos:
python3 -m app.app_client.hangman-client Guilherme27
python3 -m app.app_client.hangman-client gui localhost:10027

Introdução

Para realizar o trabalho proposto, escolhemos a linguagem Python pelos seguintes motivos: foi a recomendada no enunciado, fácil uso do protocolo TCP necessário para realizar o exercício e é uma linguagem a qual temos bastante familiaridade.

Também como recomendado, dividimos o projeto em 2 programas principais, um programa para lidar com o lado do servidor, e outro programa para lidar com o lado do cliente.

Já a estrutura do projeto, foi dividida em 4 módulos principais:

- models - Responsável por centralizar as classes utilizadas no desenvolvimento dos programas.
- app_client - Responsável por centralizar os arquivos do programa do lado do cliente.
- app_server - Responsável por centralizar os arquivos do programa do lado do servidor.
- debug - Módulo de desenvolvimento onde é possível ativar/desativar prints de debug

Tanto o módulo app_client quanto o módulo app_server, têm um arquivo de entrada principal (hangman-client.py e hangman-server.py, respectivamente), onde cada um chama o método run_game(), presente no arquivo run.py de cada módulo, que gerencia a execução do programa, se apoiando nos arquivos e métodos do sub-módulo auxiliar (client ou server, dependendo do programa a ser executado.

```
app
├── app_client
│   └── client -> biblioteca centralizadora de arquivos e métodos para execução do cliente
├── app_server
│   └── server -> biblioteca centralizadora de arquivos e métodos para execução do servidor
└── models
└── debug.py
```

Módulo *models*

A fim de tornar o desenvolvimento mais produtivo e manter a qualidade do código em todos os trechos do projeto, foram criadas classes base para auxílio na criação do programa. Tais classes facilitaram o desenvolvimento pois possibilitaram uma clara documentação, facilitando a leitura do código, permitindo melhor manipulação dos dados tratados nos programas, e também padronizando e facilitando o acesso das mensagens usadas no protocolo proposto.

Foram criadas 4 grupos de classes principais: Player, classes de GameState (ServerGameState e ClientGameState), Message (classe pai de ServerMessage e ClientMessage), e Error. Isso elevou a qualidade e a produtividade do código por vários motivos:

- Documentação e Legibilidade - As classes foram criadas com documentação clara, facilitando a leitura do código e o entendimento dos dados manipulados em todos os trechos do projeto.
- Acesso Direto e Padronizado aos Dados:
    - A criação de objetos permitiu o acesso direto aos campos, isso eliminou a necessidade de manipular dicionários inseguros, algo muito propenso a erros ao progamar em Python.
    - Separação de Estado: Criar classes separadamente garantiu que cada lado (servidor e cliente) rastreasse apenas as informações relevantes para sua perspectiva.
- Padronização e Reutilização do Protocolo (Message):
    - A classe base Message e suas herdeiras (ServerMessage, ClientMessage) padronizaram o formato de todas as mensagens. Isso facilitou o acesso e evitou erros de digitação ao longo do código, tanto para mensagens fixas quanto para mensagens que geram strings com dados.
    - A classe Error centralizou todos os códigos de erro do protocolo, garantindo que a tratativa de erros no código fosse consistente e reutilizável.

Classe Player

Para formalizar e padronizar o elemento do jogador (*Player*), foi criada uma simples classe, que contém o nome do jogador, seu endereço e seu socket de conexão com o servidor.

Essa padronização permitiu que as funções do servidor (como as de envio e remoção de jogadores) manipulassem o Player como uma entidade única, simplificando a lógica de controle de turnos e comunicação.

```
5  @dataclass
6  class Player:
7
8      socket: socket
9      name: str
10     address: str
11
```

app.models.Player

Classes GameState

Também para centralizar o controle de fluxo de jogo, foram criadas classes de estado de jogo, que ficam responsáveis por armazenar os dados importantes para cada lado de cada partida, a classe ServerGameState, para o servidor, e a classe ClientGameState, para o cliente.

A separação das classes entre servidor e cliente permitia que cada lado só tivesse a visibilidade do que era necessário para a execução, além de implementar uma camada a mais de segurança e integridade do jogo, pois, por exemplo, o jogador que não é mestre só saberá qual a palavra ao fim do jogo, caso acerte ou perca todas as vidas.

```python
@dataclass
class ServerGameState:

    word: str
    '''Palavra a ser advinhada pelos jogadores'''
    all_players: List[Player]
    '''Lista de todos os jogadores conectados'''
    master_player: Player
    '''Jogador Mestre'''
    lives: int = 7
    '''Vidas do jogo. Se chegar em 0, o jogo acaba'''
    guesses: List[str] = field(default_factory=list)
    '''Lista de palpites realizados pelos jogadores'''

    word_array: List[str] = field(init=False)
    '''Palavra a ser advinhada em forma de vetor'''
    word_progress: List[str] = field(init=False)
    '''Progresso dos palpites até a palavra a ser advinhada. Inicia apenas com "_"'''
    common_players: List[Player] = field(init=False)
    '''Lista de jogadores comuns (Não Mestre)'''
```

app.models.GameState

```python
@dataclass
class ClientGameState:

    lives: int
    '''Vidas do jogo. Se chegar em 0, o jogo acaba'''
    word_length: int
    '''Tamanho da palavra a ser advinhada'''
    is_master: bool
    '''Sinalizador de se o jogador é Mestre'''

    guesses: List[str] = field(init=False, default_factory=list)
    '''Lista de palpites errados realizados pelos jogadores'''
    word_progress: List[str] = field(init=False)
    '''Progresso dos palpites até a palavra a ser advinhada. Inicia apenas com "_"'''
```

app.models.GameState

Classe Message

Para centralizar o sistema de envio e recebimento de mensagens, foi criada a classe base Message, que contém os métodos de envio de mensagem, já formatada para o protocolo (com *terminator* \r\n), e de recebimento de mensagem, que lê a mensagem recebida em porções de bytes.

```python
class Message:

    @staticmethod
    def send_message(socket_end: socket, message: str):

        # Cooldown para envio de mensagem
        time.sleep(0.3)

        # Formata a mensagem a ser enviada
        terminator = "\r\n"
        full_message = (message + terminator).encode('ascii')

        # Envia a mensagem
        socket_end.sendall(full_message)

    @staticmethod
    def receive_message(socket: socket):

        # Inicia uma sequencia de bits vazia
        buffer = b''
        terminator = b'\r\n'
        # Leitura de s em porções
        while True:

            # Lê uma porção dos dados.
            data = socket.recv(1024)
            # Conexão encerrada pelo cliente ou erro de rede
            if not data: return None

            # Adiciona a porção ao buffer
            buffer += data
            # Verifica se o terminador está no buffer
            if terminator in buffer:
                # Limpa para retornar apenas a mensagem
                message_end_index = buffer.find(terminator)
                message = buffer[:message_end_index].decode('ascii')

                return message
```

A partir de herança foram criadas as classes ServerMessage e ClientMessage. Cada uma dessas classes armazena as mensagens que cada lado pode enviar, em string ou método que gera uma string, pois algumas mensagens carregam dados e não são fixas, e também possuem os métodos de envio e recebimento de mensagens personalizado para cada caso.

Essa separação e limitação de mensagens a serem enviadas foi muito útil, pois um dos maiores problemas era padronizar o formato das mensagens, sem precisar criar strings a cada vez que fosse escrever o envio de uma mensagem, isso acaba evitando erros de digitação.

As classes de mensagem do servidor e cliente estão expostas na próxima página.

- ServerMessage

```python
class ServerMessage(Message):

    OK = "OK"
    STANDBY = "STANDBY"
    MASTER = "MASTER"
    @staticmethod
    def NEWGAME(lives, word_len): return f"NEWGAME {lives} {word_len}"
    YOURTURN = "YOURTURN"
    @staticmethod
    def STATUS(lives, state, player_name, guess): return f"STATUS {lives} {state} {player_name} {guess}"
    @staticmethod
    def GAMEOVER(result, player, word): return f"GAMEOVER {result} {player} {word}"

    @classmethod
    def send_message_to_player(cls, player: Player, message: str):
        """Função auxiliar para enviar uma mensagem a um jogador."""

        try:

            cls.send_message(player.socket, message)

            print_debug(f"Enviei a mensagem para o jogador {player.name}:\n{message}")

        except:
            print(f"Aviso: Não foi possível enviar mensagem para o jogador {player.name}")

    @classmethod
    def send_message_to_all_players(cls, players: List[Player], message: str):
        """Função auxiliar para enviar uma mensagem a todos os jogadores."""
        for player in players: cls.send_message_to_player(player, message)

    @classmethod
    def receive_message_from_player(cls, player: Player) -> str:
        """Função auxiliar para receber uma mensagem de um jogador."""

        response = cls.receive_message(player.socket)

        print_debug(f"Recebi a mensagem do jogador {player.name}:\n{response}")

        return response
```

app.models.Message

- ClientMessage

```python
class ClientMessage(Message):

    @staticmethod
    def NEWPLAYER(player_name): return f"NEWPLAYER {player_name}"
    @staticmethod
    def WORD(word): return f"WORD {word}"
    @staticmethod
    def GUESS(type, guess): return f"GUESS {type} {guess}"

    @classmethod
    def send_message_to_server(cls, client_socket: socket, message: str, raise_exception: bool = False):
        """Função auxiliar para enviar uma mensagem ao servidor."""
        try:

            cls.send_message(client_socket, message)

            print_debug(f"Enviei a mensagem:\n{message}")

        except:
            print(f"Aviso: Não foi possível enviar mensagem para o servidor!\nMensagem: '{message}\n'")
            if raise_exception: raise

    @classmethod
    def receive_message_from_server(cls, client_socket: socket):
        """Função auxiliar para receber uma mensagem do servidor."""

        response = cls.receive_message(client_socket)

        print_debug(f"Recebi a mensagem:\n{response}")

        return response
```

app.models.Message

Classe Error

Também a fim de centralizar os possíveis erros do protocolo, foi criada uma classe apenas de strings, bem documentadas com o propósito do erro, facilitando a identificação de quando cada erro deve ser usado.

```python
class Error:
    '''Classe para centralização de erros.'''

    INVALID_FORMAT = "ERROR INVALID_FORMAT"
    '''mensagem de erro enviada sempre que houver erro de formatação da mensagem recebida'''

    INVALID_MASTER_MESSAGE = "ERROR INVALID_MASTER_MESSAGE"
    '''mensagem enviada para todos os jogadores caso o jogador mestre não envie uma palavra válida (não respondeu com mensagem WORD, ou a palavra estava ausente, ou a palavra contém caracteres inválidos'''

    UNEXPECTED_MESSAGE = "ERROR UNEXPECTED_MESSAGE"
    '''mensagem de erro enviada sempre que a mensagem recebida não for uma mensagem de erro, mas não for um dos tipos de mensagem esperada pelo protocolo naquele ponto de execução.'''

    INVALID_PLAYER_NAME = "ERROR INVALID_PLAYER_NAME"
    '''mensagem de erro enviada pelo servidor ao cliente se o nome fornecido pela mensagem NEWPLAYER for inválido. O nome é considerado inválido se estiver vazio, se contiver espaços, ou se contiver caracteres não alfanuméricos.'''

    NOT_ENOUGH_PLAYERS = "ERROR NOT_ENOUGH_PLAYERS"
    '''Mensagem de erro enviada pelo servidor ao jogador mestre caso não haja jogadores comuns restantes para continuar o jogo.'''

    ALREADY_GUESSED = "ERROR ALREADY_GUESSED"
    '''Mensagem de erro enviada pelo servidor se a mensagem GUESS do cliente contiver um palpite já enviado anteriormente (por qualquer jogador). A conexão com entre cliente e servidor não é encerrada.'''

    INVALID_LETTER = "ERROR INVALID_LETTER"
    '''Mensagem de erro enviada pelo servidor se a mensagem GUESS LETTER do cliente contiver um caractere inválido. A conexão com entre cliente e servidor não é encerrada.'''

    INVALID_WORD_LENGTH = "ERROR INVALID_WORD_LENGTH"
    '''Mensagem de erro enviada pelo servidor se a mensagem GUESS WORD do cliente contiver uma palavra tamanho diferente da palavra a ser adivinhada. A conexão com entre cliente e servidor não é encerrada.'''

    QUIT = "QUIT"
    '''Esta mensagem não é exatamente um erro, mas pode ser enviada pelo cliente ao servidor para indicar que o jogador deseja se desconectar. O servidor deve responder com OK e encerrar a conexão.'''
```

app.models.Error

Módulo *app_server*

Como exposto anteriormente, o módulo *app_server* tem um ponto de entrada (hangman-server.py) que chama a execução principal do programa, lidando com erros de execução e encerramento de execução pelo usuário.

O fluxo principal pelo lado do servidor se dá como:

1. Verificação de entradas para execução do programa (quantidade de jogadores e porta).
2. Inicialização do socket.
3. Lida com a conexão dos jogadores.
4. Escolhe o jogador mestre e recebe a palavra do mesmo.
5. Dos jogadores que não são mestre, recebem palpites tentando adivinhar a palavra, por letras ou a palavra completa.
6. Pega palpites até os jogadores acertarem ou as vidas se esgotarem, informando os jogadores a cada palpite para que possam atualizar seus estados de jogo.
7. Fecha as conexões com os jogadores de forma segura.
8. Inicia uma nova partida (volta ao ponto 2).

Tudo isso é feito com o apoio da biblioteca *server* presente no módulo, com os seguintes arquivos:

- inputs.py
  - Responsável pela verificação de argumentos passados na linha de comando para execução do programa
- socket_util.py
  - Responsável por fazer o setup do socket TCP
- game_flow.py
  - Responsável pelo gerenciamento do fluxo de jogo
  - Lida com início, finalização, verificar se o jogo continua e, se for o caso, abortar o jogo
  - Também lida com saída de jogadores, além de tratar caso não hajam mais jogadores comuns
- players.py
  - Responsável por fazer o primeiro contato com os jogadores e firmar a conexão para realizar a partida
- master.py
  - Responsável por escolher e informar o jogador de que é o Mestre, e receber a mensagem a ser adivinhada, fazendo as devidas validações e tratativas
- guess.py
  - Responsável por lidar com palpites de jogador comum
  - Valida os palpites de letra ou palavra, além de processar e atualizar o estado do jogo

Módulo *app_client*

Como exposto anteriormente, o módulo *app_client* tem um ponto de entrada (hangman-client.py) que chama a execução principal do programa, lidando com erros de execução e encerramento de execução pelo usuário.

O fluxo principal por parte do cliente se dá como:

1. Verificação de entradas para execução do programa (nome do jogador, ip e porta).
2. Conexão do socket com o servidor, enviando os dados do jogador.
3. Aguarda o início da partida.
4. Se for o caso, recebe a mensagem que é o Mestre, e envia a palavra ao servidor.
5. Se não for o mestre, recebe a mensagem de que é a sua vez de dar o palpite, e envia o palpite ao servidor
6. Recebe o status da partida para que possa atualizar seu estado de jogo.
7. Se for o caso, recebe a mensagem de fim de jogo, informa ao usuário, e encerra a conexão de forma segura com o servidor.

Tudo isso é feito com o apoio da biblioteca *client* presente no módulo, com os seguintes arquivos:

- inputs.py
  - Responsável pela verificação de argumentos passados na linha de comando para execução do programa.
- server_conn.py
  - Responsável por estabelecer a conexão do socket TCP com o servidor.
- game_flow.py
  - Responsável pelo gerenciamento do fluxo de jogo
  - Lida com espera de início, início, finalização e, se for o caso, abortar o jogo
- master.py
  - Lida com o setup do mestre e input de palavra a ser adivinhada pelo usuário, lidando com os possíveis erros
- turn.py
  - Lida com o turno do jogador
  - Recebe e valida o palpite, tratando os possíveis erros
  - Também lida com o caso do jogador querer sair da partida
- status.py
  - Atualiza o estado atual do jogo e informa ao usuário

Observações

Tratativas de erro robustas

A fim de tornar os programas resistentes a erros e cenários inesperados, basicamente para cada interação entre cliente e servidor foi implementada uma tratativa de erro bem robusta e informativa, sempre tentando alertar o outro lado de que algo não esperado havia ocorrido. A tratativa de erros engloba tanto erros recuperáveis quanto erros que necessitam que o jogo seja abortado, e ainda que o jogo precisasse ser abortado, é feita a tentativa de informar o outro lado do que ocorreu, para que possa abortar seu jogo de forma segura também.

Os principais pontos de atenção para as tratativas foram os de casos de perda de conexão, mensagens mal formatadas e mensagens inesperadas, como proposto no enunciado do exercício, fazendo bom uso da classe Errors criada especialmente para facilitar a documentação e reuso dos erros disponíveis no protocolo no fluxo do código.

Tratamento de Desconexão Silenciosa

Ambos os lados têm a capacidade de detectar e tratar a perda de conexão (o socket.recv() retorna None) e usa uma lógica defensiva, seja para remover o Player inativo da lista de jogadores e ajustar o índice de turnos, seja como o cliente ter como lidar com o encerramento do jogo e informar ao usuário, permitindo que o jogo continue ou aborte de forma controlada.

Retry em palavra de mestre e palpite

Muito atrelado ao tratamento de erros robusto, foi implementado no código a possibilidade de refazer o input tanto da palavra escolhida pelo mestre quanto de palpites enviados pelos jogadores.

Caso fosse identificado algum ponto que invalidasse o input, se possível, era dada a possibilidade de uma nova tentativa ao usuário, e, no caso do jogador mestre por exemplo, caso não fosse possível tratar, o jogo era encerrado de forma segura, e no caso de algum palpite inválido que passou despercebido pelo lado do cliente, e o servidor identificasse, a informação sobre era dispersa e o jogador perdia a vez, não bloqueando o fluxo do jogo.

Sockets reutilizáveis e configurações extras

Para garantir a estabilidade, baixo delay e resiliência do sistema, diversas configurações foram aplicadas diretamente aos sockets TCP, tanto no lado do Servidor quanto no lado do Cliente.

- Reutilização de Endereço (SO_REUSEADDR) - Com a finalidade de eliminar o erro de "endereço já em uso" (Address already in use) que ocorre após o encerramento de um socket, não bloqueando o fluxo de testes e execução dos programas.
- Desativação do Algoritmo de Nagle (TCP_NODELAY) - Com a finalidade de reduzir a latência e garantir que as mensagens de controle do protocolo sejam enviadas imediatamente, sem serem agrupadas pelo sistema operacional.

Cooldown de envio de mensagens

Um problema que tivemos no momento de testes é que, apesar do código estar estruturado corretamente, algumas mensagens não chegavam ao cliente, por conta de um envio sequencial muito rápido de mensagens (por exemplo, um YOURTURN era enviado imediatamente após um STATUS, e o cliente acabava não tendo tempo para processar, e perdia o YOURTURN, impedindo o jogo de seguir).

Para solucionar isso, foi implementado um cooldown de 300ms no envio de cada mensagem, garantindo que uma não vá se sobrepor a outra.

Isso acabou também melhorando a UX (*User Experience*) do programa, pois os passos tinham mais tempo para serem lidos.

Dificuldade para verificar se alguém saiu na etapa de conexão

Um entrave que não conseguimos resolver da melhor forma possível foi o caso de algum jogador se desconectar no momento em que a conexão de outros jogadores estivesse sendo realizada.

Talvez se o protocolo permitisse algumas trocas a mais de mensagens nesse momento de conexão, ou até mesmo usar o STANDBY diversas vezes pudesse funcionar, mas não encontramos uma forma de fazer isso que não fosse muito complexa.

Porém, no fim das contas, tornamos o restante do código robusto o suficiente para que essa questão pudesse ser ignorada e resolvida de forma passiva com o decorrer do jogo, por exemplo, o servidor identifica a saída do(s) jogador(es) ao início do jogo, mas o jogo acaba ficando com a quantidade de jogadores inicial diferente da proposta na inicialização do programa do servidor na execução pela linha de comando.

Modo Debug

Outra dificuldade do processo de desenvolvimento e testes foi a baixa visibilidade da troca de mensagens entre o servidor e os clientes, então para isso, foi criado um mini-módulo de debug, com a opção de ligar e desligar, com um método *print_debug*, onde só seria feito o print caso o módulo de debug estivesse ligado.

O print debug então foi implementado nas classes ServerMessage e ClientMessage, para termos total visibilidade das mensagens trocadas, além de inserir esse prints em outros pontos importantes do código.

Processo de Testes

O código foi testado de forma muito simples e direta, fazendo muito bem o uso do modo de debug implementado. Eram sempre abertos 4 terminais, um para o servidor e 3 para jogadores, e simulando as rodadas de forma manual.

Essa escolha foi feita pois queríamos ter total controle de cada passo da execução, podendo forçar cenários de erro ou não esperados.

Perguntas propostas no enunciado

- O que seria necessário modificar para permitir palavras com hífen?

Para que fosse possível palavras desse tipo, seria necessário tratar o hífen não como um caractere a ser adivinhado, mas sim um caractere especial, que seria revelado aos jogadores logo no início da partida, sendo uma regra a ser tratada na validação da palavra escolhida pelo mestre.

De forma efetiva, o hífen seria apenas visual, e não influenciaria na lógica de palpites, por exemplo:

- ➔ Palavra: guarda-chuva
- ➔ O que apareceria para os jogadores comuns: _ _ _ _ _ - _ _ _ _ _
- ➔ E no fundo, o servidor poderia lidar como: guardachuva

Basicamente só seria necessário armazenar a posição do caractere especial, isso daria margem para palavras de Mestre com espaço e/ou outros caracteres especiais.

- O que acontece se mais jogadores do que esperado tentarem se conectar ao servidor?

O código desenvolvido é suficientemente robusto para lidar com o caso, pois no momento de inicialização dos jogadores, só são aceitos o número especificado de jogadores, então caso algum a mais tentasse se conectar, ele iria para fila de espera do socket.

Por questões de design de jogo e evitar ter que realizar tratativas complexas para o caso, o socket é resetado a cada partida, limpando essa fila de espera, ou seja, só são aceitos os X primeiros jogadores, os que chegarem depois perdem a vez no socket de forma permanente.

- Como você melhoraria o protocolo?

Como citado anteriormente, o primeiro ponto seria facilitar a tratativa de desconexão de algum player no momento de conexão de todos os jogadores, talvez inserindo um novo tipo de mensagem reservado para tal fluxo. Isso dentre outros problemas de verificação, poderiam ser resolvidos com mensagens de verificação de conexão, como um PING/PONG.

Outro ponto seria que, levando em consideração não só as mensagens do protocolo, mas também o design de jogo proposto pelo enunciado, é que poderia ser possível a opção de não resetar os jogadores a cada partida, forçando criar uma nova conexão com o servidor. Seria interessante que, na inicialização do servidor, houvesse a escolha do modo de jogo, com o reset ou não dos jogadores.