Documentação de Encerramento do Projeto: Comunicação Paralela e/ou Cliente-Servidor

Autor: Felipe Tirone

Local: UNIC Beira Rio

Data de Encerramento: 25 de Novembro de 2025

Este artigo formaliza o encerramento do projeto individual de **Comunicação Paralela e/ou Cliente-Servidor**. O projeto teve como objetivo aplicar conceitos de arquitetura de sistemas distribuídos e concorrência, e esta documentação resume os resultados alcançados, as lições aprendidas e o status final do código.

1. Resumo Executivo e Objetivos

O projeto visou o desenvolvimento de uma aplicação que demonstrasse um modelo de comunicação em sistemas distribuídos, com foco em uma das seguintes arquiteturas: **Comunicação Paralela (Multithreading/Multiprocessing)** ou **Comunicação Cliente-Servidor (Sockets)**. O objetivo principal foi consolidar a compreensão sobre como recursos são compartilhados e como a sincronização é gerenciada em ambientes concorrentes.

Modelo Adotado: *(Preencher com o modelo escolhido e a tecnologia/linguagem utilizada. Exemplo: Comunicação Cliente-Servidor utilizando Python e a biblioteca `socket`)*

Categoria	Detalhes
Arquitetura	Comunicação Cliente-Servidor
Tecnologias/Linguagens	Python (`socket` e `threading`)
Função Principal	Servidor de eco (Echo Server) capaz de atender múltiplos clientes simultaneamente.

2. Resultados Alcançados e Funcionalidades

O projeto foi concluído com sucesso, implementando as seguintes funcionalidades essenciais:

- **Implementação da Comunicação Base:** Criação e configuração de *sockets* para a troca de dados[1].
    - **Servidor:** Configurado para ouvir em uma porta específica (Ex: porta 12345).
    - **Cliente:** Configurado para se conectar ao endereço IP e porta do servidor.
- **Concorrência/Paralelismo:** Utilização de *threads* ou *processos* para garantir que o Servidor possa manipular múltiplas conexões de clientes simultaneamente, sem bloquear a execução[2].
- **Protocolo Simples de Dados:** Definição de uma estrutura básica para o envio e recebimento de mensagens entre cliente e servidor (Ex: mensagens formatadas em JSON ou strings simples).
- **Testes de Estresse:** O sistema foi testado com *X* clientes simultâneos, demonstrando estabilidade na gestão das requisições concorrentes.

3. Detalhes de Implementação (Código e Estrutura)

A estrutura do projeto foi organizada em arquivos específicos para cada componente.

Estrutura de Arquivos:

```
/projeto_comunicacao_paralela
├── server.py             # Código do lado do Servidor.
├── client.py             # Código do lado do Cliente.
├── README.md             # Documentação de como rodar o projeto.
├── requirements.txt      # Lista de dependências (se houver).
└── .gitignore            # Arquivo para ignorar arquivos
desnecessários no GIT.
```

Trecho Crítico de Código (Exemplo de Thread do Servidor):

O trecho a seguir ilustra o gerenciamento de uma nova conexão de cliente pelo Servidor, utilizando *threading* (Comunicação Paralela) para manter o Servidor principal ativo e responsivo:

Python

```
# Pseudocódigo demonstrando a aceitação e o gerenciamento de clientes
def handle_client(conn, addr):
    print(f"Nova conexão de {addr}")
    # Lógica de recebimento e processamento de dados
```

```
def main_server():
    server_socket = socket.socket(AF_INET, SOCK_STREAM)
    server_socket.bind((HOST, PORT))
    server_socket.listen(5)

    while True:
        conn, addr = server_socket.accept()
        # Inicia uma nova thread para gerenciar o cliente
        thread = threading.Thread(target=handle_client, args=(conn,
addr))
        thread.start()
        print(f"Clientes ativos: {threading.active_count() - 1}")
```

4. Lições Aprendidas e Conclusão

O desenvolvimento deste projeto proporcionou uma compreensão aprofundada dos desafios e soluções inerentes à comunicação em sistemas distribuídos:

- **Gerenciamento de Concorrência:** A necessidade de utilizar *threads* ou mecanismos de sincronização (como *locks*) para evitar condições de corrida (*race conditions*) e garantir a integridade dos dados compartilhados (se aplicável ao projeto).
- **Manipulação de Erros:** A importância de implementar o tratamento de exceções (Ex: perda de conexão, dados corrompidos) para tornar o sistema robusto e resiliente.
- **Estrutura de Sockets:** O ciclo de vida de um socket (criação, *bind*, *listen*, *accept* no servidor; *connect* no cliente) e a diferença entre as operações bloqueantes e não bloqueantes.

Conclusão: O projeto demonstrou com sucesso a aplicação prática de arquiteturas de comunicação, validando a capacidade de construir sistemas distribuídos funcionais.

5. Próxima Etapa: Atualização do Repositório GIT

Com a documentação de encerramento completa e o código finalizado e testado, o aluno deverá agora realizar o **update** (commit e push) do repositório individual do GIT.

O repositório deve conter:

1. O código-fonte finalizado (`server.py`, `client.py`, etc.).
2. Este artigo de documentação de encerramento do projeto.
3. Um arquivo `README.md` atualizado com instruções de uso.

Comando de Exemplo para *Update*:

Bash

```
git add .
git commit -m "Encerramento do Projeto de Comunicação
Paralela/Cliente-Servidor - Documentação Final"
git push origin main
```