UNIVERSIDADE DE SÃO PAULO

ESCOLA DE ARTES, CIÊNCIAS E HUMANIDADES

PROGRAMA DE GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

MARIANA BORGES ARAUJO DA SILVA - 14596342

PEDRO SERRANO BUSCAGLIONE - 14603652

RELATÓRIO DO PRIMEIRO EXERCÍCIO PROGRAMA DA DISCIPLINA DE DESENVOLVIMENTO DE SISTEMAS DE INFORMAÇÃO DISTRIBUÍDOS

SP - São Paulo, Brasil

2025

MARIANA BORGES ARAUJO DA SILVA - 14596342

PEDRO SERRANO BUSCAGLIONE - 14603652

RELATÓRIO DO PRIMEIRO EXERCÍCIO PROGRAMA DA DISCIPLINA DE DESENVOLVIMENTO DE SISTEMAS DE INFORMAÇÃO DISTRIBUÍDOS

Relatório do trabalho de Desenvolvimento de Sistemas de Informação Distribuídos apresentado à Escola de Artes, Ciências e Humanidades da Universidade de São Paulo.

Prof. Dr. Renan Cerqueira Afonso Alves

São Paulo

2025

MAIORES DIFICULDADES ENFRENTADAS

As maiores dificuldades encontradas durante a implementação do código foram implementar a conexão entre os peers utilizando sockets, a funcionalidade de obter_peers, a função de processar a conexão, que requereu bastante tempo e ajustes e a arquitetura do código, o início dele, que precisou de tempo e bastantes tentativas para ser aperfeiçoado no melhor formato com as classes e métodos certos.

PARADIGMA DE PROGRAMAÇÃO: ESCOLHA E MOTIVAÇÃO

O principal paradigma do código é, majoritariamente a Programação Orientada a Objetos (POO), pois a estruturação do programa é feita por meio de divisão em três classes principais, sendo elas `class Clock`, `class Peer` e `class Mensagem`, onde cada classe tem seus próprios comportamentos e atributos encapsulados. O código também utiliza o conceito de concorrência com threads, o que permite que múltiplas tarefas sejam executadas ao mesmo tempo.

As principais características da POO são o **encapsulamento**, que é evidenciado pela classe Clock que protege seu estado interno e utiliza um lock para garantir consistência durante a concorrência entre as threads, a **abstração** evidenciada pela classe Mensagem que abstrai a construção e análise de mensagens trocadas entre os peers, de forma a ocultar os detalhes da implementação, a **modularização**, evidenciada pela divisão do programa em três classes principais, sendo elas `class Clock`, `class Peer` e `class Mensagem`, onde cada classe desempenha seu respectivo papel no código, o que facilita a manutenção e a futura extensão de suas funcionalidades e a **composição**, evidenciada pela classe Peer que armazenam as informações de conexão e interage com as classes `Clock` e `Mensagem`, o que permite uma estrutura modular. O código não faz o uso de herança, pois não há subclasses que derivam das classes principais, mas utiliza o **polimorfismo**, pois as classes interagem entre si de forma padronizada.

Em relação a concorrência empregada para lidar com as múltiplas conexões entre os peers, essa funcionalidade é possibilitada pelo uso da `threading.Thread()`, o que permite que cada nova conexão seja processada de forma independente. A aceitação das conexões é feita pela função

`aceitar_conexoes()`, que é responsável por criar uma nova thread para cada conexão estabelecida que age de forma independente. Além da POO, o código também utiliza alguns paradigmas procedurais, como as funções independentes que manipulam arquivos e listam peers (`listar_peers()` e `listar_arquivos()`), uso de input() e print() para interação com o usuário e a manipulação direta das listas e estruturas de controle sem o encapsulamento em objetos.

DIVISÃO DO PROGRAMA EM THREADS

A divisão do programa em threads foi feita para permitir que o os peers possam realizar várias operações ao mesmo tempo, como escutar conexões com outros peers ao mesmo tempo que interage com o usuário, de forma a não bloquear a execução principal.

Especificação da divisão

`threading.Thread()`: aceita as novas conexões entre os peers. Cada vez que um peer se conecta, uma nova thread é criada para processar a conexão:

```
183  def aceitar_conexoes(servidor, clock, lista_vizinhos, diretorio_compartilhado):
184      while True:
185          conexao, endereco = servidor.accept()
186          thread = threading.Thread(target=processar_conexao, args=(conexao, endereco, clock, lista_vizinhos, diretorio_compartilhado))
187          thread.start()
```

O loop mantém o servidor pronto para aceitar conexões, `servidor.accept()` aguarda um novo peer se conectar. Toda vez que uma nova conexão é criada, uma nova thread executa a função `processar_conexao()`, o que garante que cada conexão seja tratada separadamente, sem interromper o servidor.

`iniciar_servidor()`: quando o peer é inicializado, essa função cria uma thread para inicializar `aceitar_conexoes()`.

```
189  def iniciar_servidor(servidor, clock, lista_vizinhos, diretorio_compartilhado):
190      thread_servidor = threading.Thread(target=aceitar_conexoes, args=(servidor, clock, lista_vizinhos, diretorio_compartilhado))
191      thread_servidor.daemon = True  # Termina quando o programa principal encerra
192      thread_servidor.start()
193      return thread_servidor
```

`thread_servidor.daemon = True` define a thread como daemon, garantindo que ela termine automaticamente quando o programa principal for encerrado, `thread_servidor.start()` inicia a thread que executa `aceitar_conexoes()` e `threading.Lock` evita condições de corrida ao atualizar o relógio lógico.

```python
class Clock:
    def __init__(self):
        self.valor = 0
        self.lock = threading.Lock()

    def incrementar(self):
        with self.lock:
            self.valor += 1
            print(f"=> Atualizando relogio para {self.valor}")
            return self.valor

    def atualizar(self, valor_recebido):
        with self.lock:
            self.valor = max(self.valor, valor_recebido) + 1
            print(f"=> Atualizando relogio para {self.valor}")
            return self.valor
```

`self.lock = threading.Lock()` cria um bloqueio para evitar que múltiplas threads alterem self.valor ao mesmo tempo e `with self.lock:` garante que apenas uma thread por vez possa modificar o valor do relógio.

TIPO DE OPERAÇÕES UTILIZADAS PARA ENVIO E RECEBIMENTO DE DADOS (BLOQUEANTES OU NÃO)

As operações utilizadas para o envio e recebimento de dados foram bloqueantes, pois dessa forma é possível garantir a consistência da concorrência entre as threads e a pausa na execução do programa enquanto uma resposta de tentativa de conexão de um peer é aguardada. Os principais pontos onde são apresentados as operações bloqueantes são:

Recebimento de dados (recv)

No método `processar_conexao`, a seguinte linha é usada para receber dados de um peer:

```python
        try:
            with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as cliente:
                cliente.settimeout(5)
                cliente.connect((peer.endereco, peer.porta))
                cliente.sendall(mensagem.encode())
                resposta = cliente.recv(1024).decode()
```

O método recv() é bloqueante, ou seja, ele aguarda até que os dados sejam

recebidos ou até que a conexão seja fechada.

Envio de dados (`sendall`)

Em várias partes do código, o método `sendall()` é utilizado para enviar mensagens, como neste trecho:

```
90          try:
91              with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as cliente:
92                  cliente.settimeout(5)
93                  cliente.connect((peer.endereco, peer.porta))
94                  cliente.sendall(mensagem.encode())
95                  resposta = cliente.recv(1024).decode()
```

O método sendall() é bloqueante, pois ele tenta enviar todos os dados antes de permitir que o programa continue a execução.

Uso de settimeout()

O código usa cliente.settimeout(5), que define um tempo limite para operações de envio e recebimento. Isso impede que o programa fique indefinidamente bloqueado as oprações ao tentar se conectar a um peer que não responde.

```
90          try:
91              with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as cliente:
92                  cliente.settimeout(5)
93                  cliente.connect((peer.endereco, peer.porta))
94                  cliente.sendall(mensagem.encode())
95                  resposta = cliente.recv(1024).decode()
```

Threads para evitar bloqueios completos

Como o código utiliza **threads** para aceitar conexões e processar mensagens (`iniciar_servidor` e `aceitar_conexoes`), o bloqueio dessas operações não paralisa a execução global do programa.

ESTRUTURA DE DADOS: ESCOLHA E MOTIVAÇÃO

As principais estruturas de dados utilizadas são as `Listas (list)`, `Strings (str)`, `Inteiros (int)`, `Objetos (Classes Peer, Mensagem, Clock)`.

Detalhamento

As Listas são amplamente utilizadas no código para armazenar e gerenciar conjuntos de elementos. No código, alguns exemplos são:

- **Lista de peers (lista_vizinhos):** armazena os peers conhecidos na rede.
- **Argumentos das mensagens (argumentos):** guarda informações adicionais enviadas nas mensagens de comunicação.

```
230     lista_vizinhos = []
231     with open(arquivo_vizinhos, "r") as arquivo:
232         for linha in arquivo:
233             linha = linha.strip()
234             if linha:
235                 endereco_peer, porta_peer = linha.split(":")
236                 peer = Peer(endereco_peer, int(porta_peer))
237                 lista_vizinhos.append(peer)
238                 print(f"Adicionando novo peer {linha} status {peer.estado}")
```

```
33  class Mensagem:
34      def __init__(self, origem, clock, tipo, argumentos=None):
35          self.origem = origem
36          self.clock = clock
37          self.tipo = tipo
38          self.argumentos = argumentos or []
39
40      def construir_mensagem(self):
41          mensagem = f"{self.origem} {self.clock} {self.tipo}"
42          if self.argumentos:
43              mensagem += " " + " ".join(self.argumentos)
44          mensagem += "\n"
45          return mensagem
46
47      @staticmethod
48      def analisar_mensagem(mensagem_str):
49          partes = mensagem_str.strip().split(" ")
50
51          if len(partes) < 3:
52              raise ValueError(f"Formato inválido da mensagem recebida: '{mensagem_str}'")
53
54          try:
55              origem = partes[0]
56              clock = int(partes[1])  # <- Aqui ocorre o erro
57              tipo = partes[2]
58              argumentos = partes[3:]
59          except ValueError:
60              raise ValueError(f"Erro ao converter clock para inteiro na mensagem: '{mensagem_str}'")
61
62          return Mensagem(origem, clock, tipo, argumentos)
```

O uso de listas é apropriado porque elas permitem acesso sequencial e fácil modificação, o que é útil para gerenciar peers dinâmicos na rede.

As Strings desempenham um papel de comunicação entre os peers, pois são utilizadas para formatar e enviar mensagens entre os nós da rede. Alguns exemplos no código são:

- **Mensagens enviadas e recebidas:** As mensagens trocadas entre os peers são representadas como strings, facilitando a serialização e desserialização dos dados.
- **Endereços dos peers:** Os endereços e portas dos peers são armazenados como strings para facilitar a manipulação e exibição.

Os Inteiros são utilizados em diferentes partes do código, principalmente para:

- **Controle do relógio lógico (Clock):** Cada peer mantém um valor inteiro que representa seu tempo lógico, garantindo a ordenação correta dos eventos.
- **Portas de comunicação:** Os números inteiros são usados para representar as portas dos peers na rede.

Os objetos do código segue o paradigma da POO, encapsulando diferentes entidades em classes:

- **Peer:** Representa um nó da rede e mantém informações sobre seu estado.
- **Mensagem:** Modela as mensagens trocadas entre os peers, garantindo padronização na comunicação.
- **Clock:** Implementa um relógio lógico para sincronização de eventos.

O uso de classes favorece a modularização do código, tornando-o mais estruturado e de fácil manutenção.

CLASSES ESCOLHIDAS

As classes escolhidas foram Clock, Peers e Mensagem.

A classe Clock é responsável por controlar o relógio lógico de cada peer (os peers não partilham de um relógio local). Essa classe tem a função de manter a ordem dos eventos, evitando problemas de concorrência e de ordenação das mensagens. Os principais métodos da classe Clock são **`incrementar()`:** Aumenta o valor do relógio local e **`atualizar(valor_recebido)`** que ajusta o

relógio considerando o valor recebido de outro peer, garantindo a coerência temporal.

A Classe Peer representa cada nó na rede distribuída e mantém as informações sobre sua identidade e estado. Essa classe facilita o gerenciamento dos peers conectados, armazenando seu status conforme as iterações ocorrem. Os principais atributos da classe Peer são **`endereco`** que representa IP ou nome do host do peer, a **`porta`** Porta de comunicação e **`estado`**, que pode ser "ONLINE" ou "OFFLINE". O principal método da classe é **`atualizar_estado(novo_estado)`**, que modifica o estado do peer permitindo atualizações conforme as interações na rede são efetuadas.

A Classe Mensagem gerencia a comunicação entre os peers, criando e interpretando mensagens enviadas pela rede, essa classe padroniza a comunicação entre os peers, facilitando o envio e o processamento das mensagens na rede. Os principais atributos da classe Mensagem são **`origem`** que indica o Peer que enviou a mensagem, **`clock`**: que indica o valor do relógio lógico no momento do envio e **`tipo`**, que indica o tipo da mensagem (exemplo: "HELLO", "GET_PEERS", "PEER_LIST"). Os principais métodos são **`construir_mensagem()`** que gera uma string formatada para envio e **`analisar_mensagem(mensagem_str)`** que interpreta uma string recebida e retorna um objeto Mensagem.

TESTES

Os testes feitos foram os exemplos que estavam no enunciado do EP e alguns outros trocando os endereços, portas e a lista de vizinhos do peer iniciado, de modo a garantir que todas as funcionalidades estavam de acordo com o enunciado.

COMPILAR E EXECUTAR

Para compilar e executar o código é necessário python instalado, um folder contendo os arquivos EACHare.py, vizinhos.txt e o folder diretorio_compartilhado com alguns arquivos dentro dele, podem ser aleatórios e seguido para rodar o código é preciso rodar esse formato de linha no terminal: **python3 EACHare.py 127.0.0.1:8080 vizinhos.txt diretorio_compartilhado**, o que inicia um peer. Para

testar todas as funcionalidades é preciso iniciar mais de um peer, preferencialmente com uma lista de vizinhos diferente.