UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ARTES, CIÊNCIAS E HUMANIDADES
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

SISTEMA DE COMPARTILHAMENTO DE ARQUIVOS PEER-TO-PEER

Exercício Programa: parte 2

FILIPE BISON DE SOUZA - 14577653 - Turma 04
LUAN PEREIRA PINHEIRO - 13672471 - Turma 04

SÃO PAULO
2025

SUMÁRIO

1. PROPOSTA

1.1. Objetivo

Este projeto tem como objetivo implementar um sistema de compartilhamento de arquivos peer-to-peer simplificado chamado EACHare.

2. IMPLEMENTAÇÃO

A implementação foi feita parte em classes e parte em funções definidas no arquivo principal do projeto.

2.1. Diminuição de uso de conexões desnecessárias

Anteriormente, o envio de mensagens entre peers era sempre feito criando uma nova conexão para cada requisição, independentemente do tipo de mensagem. Isso funcionava para mensagens simples como HELLO ou BYE, mas complicava o fluxo quando esperávamos uma resposta direta, como no caso do GET_PEERS, em que o peer envia uma solicitação e espera pela lista de peers em seguida.

A dificuldade estava justamente em aproveitar a conexão já aberta para enviar a resposta. Na versão anterior, o handle_client recebia a mensagem (recv) e já encerrava o socket no finally. Isso inviabilizava manter o canal aberto até o envio da resposta. Para corrigir o problema, foi necessário reestruturar o handle_client, passando o socket para o handle_message e removendo o fechamento da conexão no send_message, garantindo que a resposta fosse tratada corretamente antes de encerrar a comunicação.

Além disso, foi criada a função send_answer, que envia a resposta diretamente pela mesma conexão aberta, evitando assim a necessidade de abrir outra conexão só para responder. No final, essa conexão é fechada manualmente após o envio da resposta, o que resolve o problema da desconexão prematura.

Essa alteração eliminou conexões desnecessárias, melhorou a eficiência e permitiu que o sistema passasse a diferenciar melhor quando deve manter ou encerrar o socket, de acordo com o tipo de mensagem.

2.2. Diminuição da limitação do número de bytes

No código anterior, usava-se um recv(1024) direto e se assumia que toda a mensagem seria recebida de uma vez. Essa suposição é arriscada, já que as

mensagens podem ser segmentadas — especialmente as maiores — e não chegar de forma completa em um único recv.

A nova lógica substitui essa abordagem fixa por um controle mais inteligente: foi introduzido um buffer que acumula os dados recebidos até encontrar um caractere de quebra de linha (\n), indicando o fim da mensagem. Com isso, garantimos que o processamento da mensagem só ocorre quando ela está de fato completa.

Além disso, trocamos o uso de send() por sendall(). Essa mudança é fundamental: o send() não garante que todos os bytes da mensagem sejam enviados de uma vez, especialmente em conexões com alta carga ou buffers pequenos. Já o sendall() tenta enviar todos os dados até o final ou lança uma exceção se falhar, o que é essencial para garantir a integridade da comunicação. Isso também melhora a robustez do envio de mensagens, que agora é mais confiável, mesmo para mensagens maiores.

2.3. Relógio de Lamport

A lógica de tempo foi feita com o relógio de Lamport, ao invés de usar um relógio puramente local, o que garante a sincronização com outros peers.

Na nova versão, foi adicionada a função update_clock(peer_clock), que é chamada ao receber uma mensagem. Essa função compara o valor do relógio local com o valor recebido na mensagem e atualiza o local com max(local, recebido) + 1. Dessa forma, garantimos que eventos causais tenham um ordenamento lógico, mesmo que ocorram em máquinas diferentes sem relógio físico sincronizado, conforme a especificação do projeto.

Além disso, antes de qualquer envio de mensagem, o relógio é incrementado com increment_clock(), mantendo a consistência temporal do peer emissor, como já era realizado na parte 1.

2.4. Alterações no gerenciamento de peers conhecidos

Diferentemente da primeira parte do EP, nesta segunda os peers são armazenados juntamente com o maior clock visto de cada um deles. Armazenar esse número foi muito fácil com a nossa implementação pois já estávamos guardando um “mysterious_number”, que era enviado sempre como “0” nas

mensagens do tipo PEER_LIST da primeira parte. Então, precisamos apenas trocar o 0 pelo clock mais recente do peer.

Cada peer tem seu estado atualizado sempre que recebemos mensagem direta dele, ou quando recebemos uma lista de peers e a informação do respectivo peer é mais recente na lista do que localmente. Para o segundo tipo, a implementação foi bem direta. Para o primeiro, fizemos algumas adaptações no código para reduzir a quantidade de código repetido, como setar o peer que está enviando a mensagem como online por padrão e apenas alterar se a mensagem for do tipo BYE.

2.5. Comando Buscar

A implementação do comando buscar pôde ser feita de forma relativamente direta. Ao selecionar a opção buscar, uma mensagem do tipo LS é enviada para todos os peers que acreditamos que estão online. Já tínhamos uma função para retornar uma lista com esses peers, então bastou iterar por ela e enviar as mensagens.

No novo código, conseguimos indicar se uma mensagem deve ou não esperar por respostas, então a implementação considerando que esperamos uma resposta do tipo LS_LIST foi bem direta. Recebemos as respostas em um vetor gerenciado por Connection e iteramos pelo vetor para separar a resposta por arquivos, tomando o cuidado de resetar o vetor toda vez que o comando buscar for solicitado.

Para enviar a resposta do tipo LS_LIST, precisamos mudar a forma como guardamos o diretório de arquivos compartilhados, para que pudéssemos enxergá-lo a partir do connection.py e não só do main.py. Além disso, bastou usar as funções da biblioteca “os” do Python para enviar as informações necessárias.

Para fazer o download de um arquivo encontrado na rede, enviamos uma mensagem do tipo DL para o peer que contém o arquivo, conforme a especificação. Utilizamos a biblioteca própria do Python para fazer as conversões dos arquivos para base 64. Do mais, a implementação foi direta, assim como a implementação da resposta do tipo FILE.

Como adicionamos mais mensagens que necessitam de respostas, fizemos uma refatoração no método handle_message da classe Connection, de modo que o código ficasse mais claro e com menos redundância. Para isso, usamos um conjunto para identificar rapidamente se uma mensagem recebida é desconhecida,

e um outro para identificar se ela é uma resposta (o que altera a impressão da mensagem recebida entre “Resposta recebida” e “Mensagem recebida”, a depender do tipo).

2.6. Decisões de projeto

Tomamos algumas decisões de projeto em relação a itens que não estavam especificados no EP.

Uma delas foi o de considerar o clock inicial de todo peer conhecido a partir do arquivo passado ao inicializar o programa como 0.

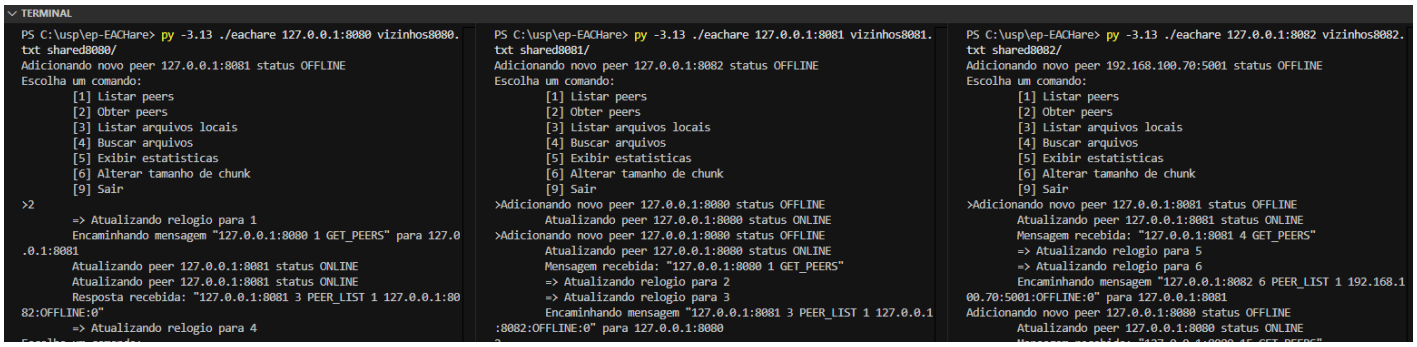
Além disso, consideramos que as respostas (FILE, LIST_PEERS, LS_LIST) somente são enviadas depois de receberem uma solicitação do respectivo peer, de modo que o código fica mais simples e legível.

3. TESTES REALIZADOS

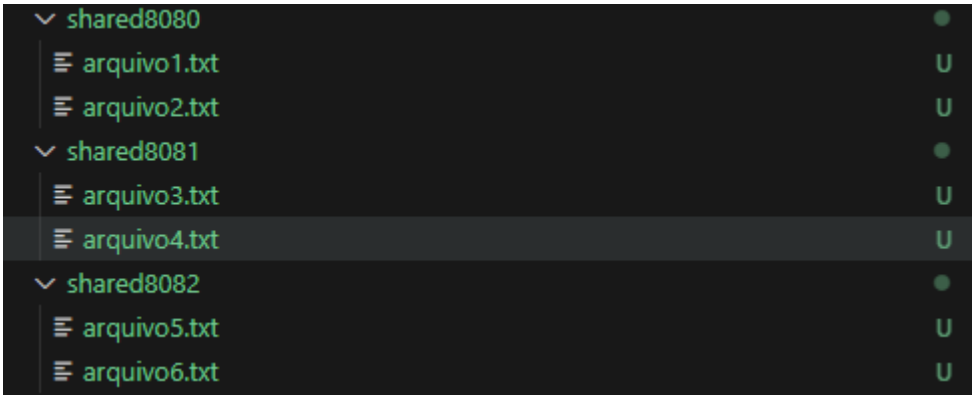
Ao longo da implementação do código, fizemos alguns pequenos testes para averiguar que cada função estava funcionando da forma que devia. Além disso, fizemos alguns testes específicos, como visto a seguir.

3.1. Re-Execução do exemplo 4

Foram iniciados 3 servidores e executando o comando para que todos soubessem que todos estão onlines.



Os arquivos estavam divididos da seguinte forma:



Após buscar os arquivos:

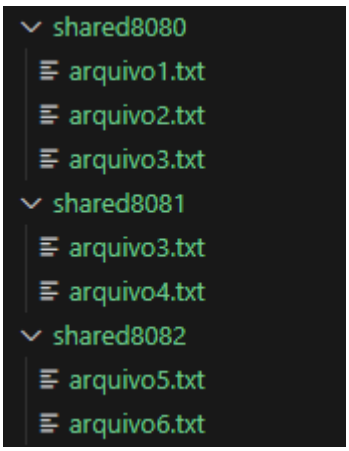
```
Escolha um comando:
        [1] Listar peers
        [2] Obter peers
        [3] Listar arquivos locais
        [4] Buscar arquivos
        [5] Exibir estatisticas
        [6] Alterar tamanho de chunk
        [9] Sair
>4
        => Atualizando relogio para 8
        Encaminhando mensagem "127.0.0.1:8080 8 LS" para 127.0.0.1:8081
        Atualizando peer 127.0.0.1:8081 status ONLINE
        Atualizando peer 127.0.0.1:8081 status ONLINE
        Resposta recebida: "127.0.0.1:8081 10 LS_LIST 2 arquivo3.txt:82 arquivo4.txt:98"
        => Atualizando relogio para 11
        => Atualizando relogio para 12
        Encaminhando mensagem "127.0.0.1:8080 12 LS" para 127.0.0.1:8082
        Atualizando peer 127.0.0.1:8082 status ONLINE
        Atualizando peer 127.0.0.1:8082 status ONLINE
        Resposta recebida: "127.0.0.1:8082 14 LS_LIST 2 arquivo5.txt:82 arquivo6.txt:98"
        => Atualizando relogio para 15
Arquivos encontrados na rede:
             Nome               | Tamanho    | Peer
        [ 0] <Cancelar>         |            |
        [ 1] arquivo3.txt       | 82         | 127.0.0.1:8081
        [ 2] arquivo4.txt       | 98         | 127.0.0.1:8081
        [ 3] arquivo5.txt       | 82         | 127.0.0.1:8082
        [ 4] arquivo6.txt       | 98         | 127.0.0.1:8082

Digite o numero do arquivo para fazer o download:
>1
```

Após a execução do download:

```
Digite o numero do arquivo para fazer o download:
>1
arquivo escolhido arquivo3.txt
        => Atualizando relogio para 16
        Encaminhando mensagem "127.0.0.1:8080 16 DL arquivo3.txt 0 0" para 127.0.0.1:8081
        Atualizando peer 127.0.0.1:8081 status ONLINE
        Atualizando peer 127.0.0.1:8081 status ONLINE
        Resposta recebida: "127.0.0.1:8081 18 FILE arquivo3.txt 0 0 MTkyLjE2OC4xMDAuNTA6NTAwMA0KM
TkyLjE2OC4xMDAuNTA6NTAwMQ0KMTkyLjE2OC4xMDAuNjA6NTAwMA0KMTkyLjE2OC4xMDAuNzA6NTAwMQ=="
        => Atualizando relogio para 19
Download do arquivo arquivo3.txt finalizado.
```

Podemos visualizar o efeito prático do download abaixo:

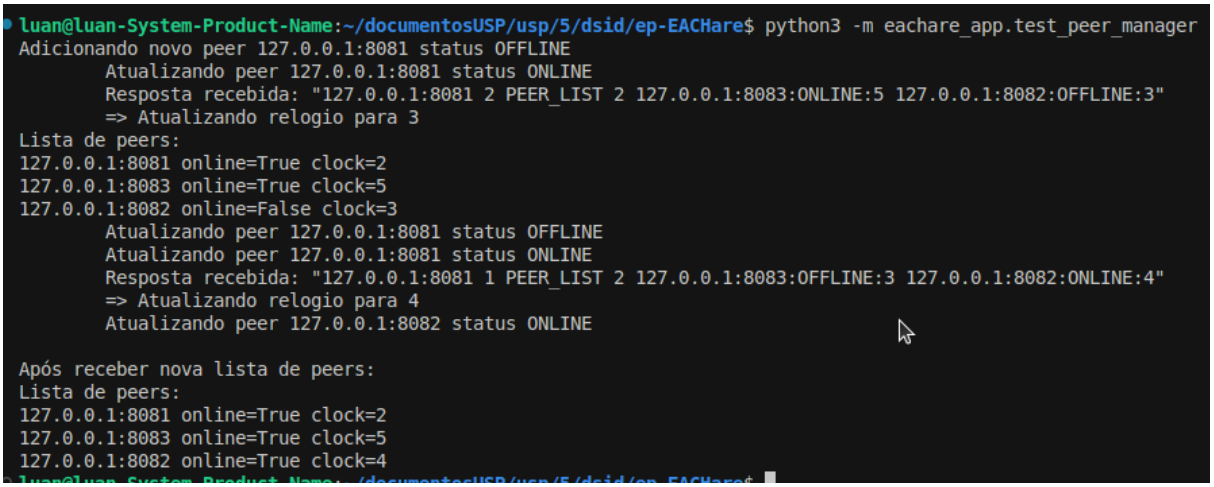


3.2. Testes unitários

Como os exemplos necessários para testar o bom funcionamento do programa para essa parte do EP se tornaram mais complexos, realizamos alguns testes unitários, especialmente para testar o gerenciamento de peers.

```
eachare_app > test_peer_manager.py > ...
20  from eachare_app.peer_manager import PeerManager
19  from eachare_app.connection import Connection
18
17  # Instancia o PeerManager e Connection
16  pm = PeerManager()
15  conn = Connection('127.0.0.1', 8080, pm)
14
13  # Simula recebimento de uma lista de peers
12  # Formato: <ip:porta> <clock> PEER_LIST <peer1> <peer2> ...
11  peer_list_message = "127.0.0.1:8081 2 PEER_LIST 2 127.0.0.1:8083:ONLINE:5 127.0.0.1:8082:OFFLINE:3"
10  conn.handle_message(peer_list_message, None)
9
8   def print_peer_list():
7       peers = pm.list_peers()
6       print("Lista de peers:")
5       for peer in peers:
4           print(f"{peer.ip}:{peer.port} online={peer.online} clock={peer.clock}")
3
2   print_peer_list()
1
21  # Simula mudança de status do 8081
1   pm.get_peer("127.0.0.1", 8081).set_offline()
2
3   # Simula recebimento de outra lista de peers
4   peer_list_message = "127.0.0.1:8081 1 PEER_LIST 2 127.0.0.1:8083:OFFLINE:3 127.0.0.1:8082:ONLINE:4"
5   conn.handle_message(peer_list_message, None)
6   print("\nApós receber nova lista de peers:")
7
8   print_peer_list()
9
```

A execução do exemplo acima, que testa o gerenciamento tanto do clock dos peers quanto de si próprio pode ser vista abaixo:



Todos os valores estão de acordo com o esperado, portanto o teste foi bem-sucedido.