

**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 3**

**FILIPPE BISON DE SOUZA - 14577653 - Turma 04  
LUAN PEREIRA PINHEIRO - 13672471 - Turma 04**

**SÃO PAULO  
2025**

## SUMÁRIO

<b>1. PROPOSTA</b>	<b>3</b>
1.1. Objetivo	3
<b>2. IMPLEMENTAÇÃO</b>	<b>3</b>
2.1. Alterar tamanho de chunk	3
2.2. Alterações no funcionamento do comando Buscar	3
2.3. Abreviação dos logs de mensagem	4
2.4. Levantamento e exibição de estatísticas	5
2.5. Decisões de projeto	6
<b>3. TESTES REALIZADOS E EXPERIMENTOS</b>	<b>6</b>
3.1. Experimento sobre a influência do tamanho de chunk	6
3.2. Experimento sobre a influência do número de peers	10

## **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. Alterar tamanho de chunk**

Função simples. Apenas altera um parâmetro de uma instância da classe Connection. Vale ressaltar que a esse ponto o tamanho do chunk ainda não está sendo usado, e seu uso se dará apenas a partir das alterações no comando Buscar, que serão explicadas a seguir.

### **2.2. Alterações no funcionamento do comando Buscar**

Para alterar o funcionamento do comando Buscar, primeiramente os dados referente aos arquivos no peer solicitante (nome, tamanho e endereço) foram salvos de forma diferente da segunda parte do EP. Agora, um dicionário é usado no lugar de uma lista, de modo que arquivos com mesmo nome e tamanho sejam armazenados juntos (cada entrada no dicionário guarda a lista de peers que o contém). Uma outra estrutura auxiliar usada nessa parte foi uma lista das chaves (nome, tamanho), para ligar a entrada do usuário (um número) a uma chave do dicionário.

A aquisição dos arquivos via rede foi drasticamente alterada. Agora, os arquivos são solicitados chunk a chunk de forma paralela entre os peers detentores daquele arquivo. Para realizar essa tarefa, usamos um modelo simplificado do modelo produtor-consumidor.

Sabendo a quantidade de chunks da busca (temos essa informação pois temos o tamanho do arquivo e o tamanho de um chunk), podemos facilmente separar os chunks sequencialmente e usar um índice para se referir a cada um deles (índices vão de 0 a num\_chunks-1). Tendo isso em vista, criamos uma thread ("worker") para cada peer, visando maximizar a quantidade de peers trabalhando ao

mesmo tempo e sem sobrecarregá-los com múltiplas conexões. Estes workers se comunicam diretamente com seus respectivos peers, enviando mensagens “DL” e coletando as respostas do tipo “FILE” em uma lista compartilhada. Quando um worker termina sua tarefa, ele observa o índice compartilhado e, se ele corresponder a um chunk existente, ele incrementa o índice compartilhado e realiza a tarefa para esse chunk.

Quando acabam as tarefas pendentes, as threads são fechadas e os resultados são ordenados para formar o arquivo completo, que é escrito em disco.

É importante ressaltar que para essa busca ser feita de forma segura, alguns locks são necessários para garantir que não tenhamos problemas de condição de corrida. Esses locks são necessários no índice compartilhado de chunks, na lista compartilhada de respostas “FILE” e nos “prints” que acontecem em cada thread.

No peer detentor do arquivo a mudança consistiu em deixar de enviar o arquivo completo e passar a enviar apenas o chunk pedido. Para encontrar o chunk, usamos um seek, que tem complexidade  $O(1)$ , garantindo que a separação do arquivo realmente seja eficiente.

### 2.3. Abreviação dos logs de mensagem

Para abreviar as mensagens e não poluir a impressão de logs, foi utilizada uma função que para o tipo de mensagem FILE realiza um processamento, limitando o conteúdo que vai aparecer a 30 caracteres, adicionando reticências para indicar que aquele não é o tamanho real do arquivo.

```
def abbreviate_message(self, message: str) -> str:
    """Encurta o conteúdo da mensagem FILE apenas para exibição."""
    parts = message.strip().split(" ")
    if len(parts) >= 4 and parts[2] == "FILE":
        # Limita o conteúdo do arquivo (última parte) a 30 caracteres
        shortened_content = parts[5][:30] + "..." if len(parts[5]) > 30 else parts[5]
        parts[5] = shortened_content
    return " ".join(parts)
return message
```

Isso resultou na seguinte impressão:

```
Encaminhando mensagem "127.0.0.1:8082 23 DL 1kb_dummy_file 256 1" para 127.0.0.1:8080
Atualizando peer 127.0.0.1:8080 status ONLINE
Atualizando peer 127.0.0.1:8080 status ONLINE
Resposta recebida: "127.0.0.1:8080 25 FILE 1kb_dummy_file 256 1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAA..."
```

## 2.4. Levantamento e exibição de estatísticas

O levantamento de estatísticas é realizado pela função `add_download_statistics` a qual tem o objetivo de registrar dados relacionados a downloads realizados no sistema. Para isso, consideramos três parâmetros principais que caracterizam cada tipo de download: o tamanho do arquivo, o tamanho dos chunks (partes em que o arquivo foi dividido) e o número de peers envolvidos no processo. Esses três parâmetros são agrupados em uma tupla que serve como chave (`'key'`) de um dicionário global chamado `download_statistics`, onde cada entrada representa um cenário específico de download, o uso da chave permite acesso rápido às estatísticas específicas, tornando a solução escalável à medida que diferentes configurações de download são testadas, já que a busca em um dicionário é de  $O(1)$ .

Além das chaves que identificam tipo de execução é passado o tempo da última execução para registrar as estatísticas (média e desvio padrão), dessa forma foi medido o tempo inicial, antes da inicialização das threads para cada peer, e final, após a ordenação dos lotes recebidos pelas threads. A medição foi realizada utilizando a biblioteca `time` padrão do python.

Ao registrar uma nova estatística, verificamos se aquele cenário já foi registrado anteriormente. Se for a primeira ocorrência, inicializamos a entrada com os valores básicos: número de amostras igual a 1, o tempo médio igual ao próprio tempo do primeiro download, e as variáveis auxiliares `'M2'` e `'stddev'` (usada para calcular o desvio padrão incrementalmente) iniciadas com zero. O campo `'M2'` armazena a soma dos quadrados das diferenças, permitindo calcular o desvio padrão sem a necessidade de armazenar todos os tempos individualmente.

Para os casos em que já existem dados registrados para aquele cenário, utilizamos o algoritmo de Welford para atualizar a média e o desvio padrão de forma incremental. Esse algoritmo é conhecido por sua eficiência numérica e simplicidade, pois evita o acúmulo de erro de ponto flutuante ao calcular a média e a variância online. Após atualizar os valores, armazenamos novamente as informações no dicionário global.

Já para a exibição foi utilizada uma função para apresentar as estatísticas de forma tabular. Para isso, formatamos a saída em colunas alinhadas que mostram o tamanho do chunk, o número de peers, o tamanho do arquivo, a quantidade de

registros coletados (`count`), o tempo médio de download e o desvio padrão associado.

## 2.5. Decisões de projeto

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

Embora pudéssemos realizar conexões persistentes durante o download, já que enviamos múltiplas mensagens para o mesmo peer em sequência; optamos por continuar usando conexões stateless porque avaliamos que os ganhos não compensariam o esforço de criar conexões stateful, visto que aumentariam muito a complexidade do programa e as chances de ocorrerem novos bugs não previstos.

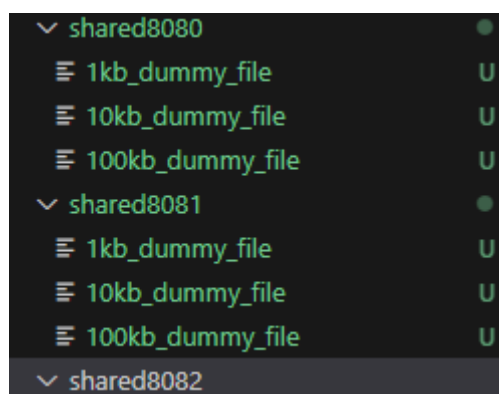
## 3. TESTES REALIZADOS E EXPERIMENTOS

Para validar a influência de alguns fatores no tempo de download foram realizados experimentos com a transferência de arquivos de diferentes tamanhos. Para gerar esses arquivos foi utilizado o comando *fsutil* do windows.

```
PS C:\usp\ep-EACHare> fsutil file createnew 1kb_dummy_file 1000
Arquivo C:\usp\ep-EACHare\1kb_dummy_file criado
PS C:\usp\ep-EACHare> fsutil file createnew 10kb_dummy_file 10000
Arquivo C:\usp\ep-EACHare\10kb_dummy_file criado
PS C:\usp\ep-EACHare> fsutil file createnew 100kb_dummy_file 100000
Arquivo C:\usp\ep-EACHare\100kb_dummy_file criado
```

### 3.1. Experimento sobre a influência do tamanho de chunk

A ideia desse experimento é iniciar três servidores, sendo que dois possuirão os mesmos arquivos, e o terceiro solicitará o download desses arquivos com chunks diferentes.



```
▼ shared8080 ●
  ≡ 1kb_dummy_file U
  ≡ 10kb_dummy_file U
  ≡ 100kb_dummy_file U
▼ shared8081 ●
  ≡ 1kb_dummy_file U
  ≡ 10kb_dummy_file U
  ≡ 100kb_dummy_file U
▼ shared8082
```

São iniciados os três servidores, e iniciada a conexão para que todos estejam conectados:

```
PS C:\usp\ep-EACHare> py -3.13 ./eachare 127.0.0.1:8080 vizinhos8080.txt shared8080/
Adicionando novo peer 127.0.0.1:8081 status OFFLINE
Escolha um comando:
[1] Listar peers
[2] Obter peers
[3] Listar arquivos locais
[4] Buscar arquivos
[5] Exibir estatísticas
[6] Alterar tamanho de chunk
[9] Sair
>2
=> Atualizando relógio para 1
Encaminhando mensagem "127.0.0.1:8080 1 GET_PEERS" 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 6 PEER_LIST 2 127.0.0.1:8082:ONLINE:3 192.168.100.70:5001:OFFLINE:0"
=> Atualizando relógio para 7
Escolha um comando:
[1] Listar peers
[2] Obter peers
[3] Listar arquivos locais
[4] Buscar arquivos
[5] Exibir estatísticas
[6] Alterar tamanho de chunk
[9] Sair
>2
=> Atualizando relógio para 8
Encaminhando mensagem "127.0.0.1:8080 8 GET_PEERS" 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 PEER_LIST 2 127.0.0.1:8082:ONLINE:3 192.168.100.70:5001:OFFLINE:0"

PS C:\usp\ep-EACHare>
PS C:\usp\ep-EACHare> py -3.13 ./eachare 127.0.0.1:8081 vizinhos8081.txt shared8081/
Adicionando novo peer 127.0.0.1:8082 status OFFLINE
Escolha um comando:
[1] Listar peers
[2] Obter peers
[3] Listar arquivos locais
[4] Buscar arquivos
[5] Exibir estatísticas
[6] Alterar tamanho de chunk
[9] Sair
>2
=> Atualizando relógio para 1
Encaminhando mensagem "127.0.0.1:8081 1 GET_PEERS" 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 3 PEER_LIST 1 192.168.100.70:5001:OFFLINE:0"
=> Atualizando relógio para 4
Escolha um comando:
[1] Listar peers
[2] Obter peers
[3] Listar arquivos locais
[4] Buscar arquivos
[5] Exibir estatísticas
[6] Alterar tamanho de chunk
[9] Sair
>Adicionando novo peer 127.0.0.1:8080 status OFFLINE
Atualizando peer 127.0.0.1:8080 status ONLINE
Mensagem recebida: "127.0.0.1:8080 1 GET_PEERS"
=> Atualizando relógio para 5
=> Atualizando relógio para 6
Encaminhando mensagem "127.0.0.1:8081 6 PEER_LIST 2 127.0.0.1:8082:ONLINE:3 192.168.100.70:5001:OFFLINE:0"

PS C:\usp\ep-EACHare>
PS C:\usp\ep-EACHare> py -3.13 ./eachare 127.0.0.1:8082 vizinhos8082.txt shared8082/
Adicionando novo peer 192.168.100.70:5001 status OFFLINE
Escolha um comando:
[1] Listar peers
[2] Obter peers
[3] Listar arquivos locais
[4] Buscar arquivos
[5] Exibir estatísticas
[6] Alterar tamanho de chunk
[9] Sair
>Adicionando novo peer 127.0.0.1:8081 status OFFLINE
Atualizando peer 127.0.0.1:8081 status ONLINE
Mensagem recebida: "127.0.0.1:8081 1 GET_PEERS"
=> Atualizando relógio para 2
=> Atualizando relógio para 3
Encaminhando mensagem "127.0.0.1:8082 3 PEER_LIST 1 192.168.100.70:5001:OFFLINE:0" para 127.0.0.1:8081
Adicionando novo peer 127.0.0.1:8080 status OFFLINE
Atualizando peer 127.0.0.1:8080 status ONLINE
Mensagem recebida: "127.0.0.1:8080 12 GET_PEERS"
=> Atualizando relógio para 13
=> Atualizando relógio para 14
Encaminhando mensagem "127.0.0.1:8082 14 PEER_LIST 2 192.168.100.70:5001:OFFLINE:0 127.0.0.1:8081:ONLINE:1" para 127.0.0.1:8080
[]
```

Pode se observar no terminal do terceiro servidor de porta 8082 que é possível visualizar todos os arquivos dos outros dois servidores:

```
Escolha um comando:
[1] Listar peers
[2] Obter peers
[3] Listar arquivos locais
[4] Buscar arquivos
[5] Exibir estatísticas
[6] Alterar tamanho de chunk
[9] Sair
>4
=> Atualizando relógio para 15
Encaminhando mensagem "127.0.0.1:8082 15 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 17 LS_LIST 3 100kb_dummy_file:100000 10kb_dummy_file:1000
0 1kb_dummy_file:1000"
=> Atualizando relógio para 18
=> Atualizando relógio para 19
Encaminhando mensagem "127.0.0.1:8082 19 LS" para 127.0.0.1:8080
Atualizando peer 127.0.0.1:8080 status ONLINE
Atualizando peer 127.0.0.1:8080 status ONLINE
Resposta recebida: "127.0.0.1:8080 21 LS_LIST 3 100kb_dummy_file:100000 10kb_dummy_file:1000
0 1kb_dummy_file:1000"
=> Atualizando relógio para 22
Arquivos encontrados na rede:
Nome | Tamanho | Peer
[ 0] <Cancelar> | | 
[ 1] 100kb_dummy_file | 100000 | 127.0.0.1:8081, 127.0.0.1:8080
[ 2] 10kb_dummy_file | 10000 | 127.0.0.1:8081, 127.0.0.1:8080
[ 3] 1kb_dummy_file | 1000 | 127.0.0.1:8081, 127.0.0.1:8080

Digite o número do arquivo para fazer o download:
```

Foram realizados os downloads de todos os arquivos cinco vezes para buscar uma média de tempo mais relevante estatisticamente.

```
Encaminhando mensagem "127.0.0.1:8082 4381 DL 100kb_dummy_file 256 389" 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 4383 FILE 100kb_dummy_file 256 389 AAAAAAAAAAAAAAAAAAAAAAAAAAAAA..."
=> Atualizando relógio para 4384
=> Atualizando relógio para 4385
Encaminhando mensagem "127.0.0.1:8082 4385 DL 100kb_dummy_file 256 385" para 127.0.0.1:8080
Atualizando peer 127.0.0.1:8080 status ONLINE
Atualizando peer 127.0.0.1:8080 status ONLINE
Resposta recebida: "127.0.0.1:8080 4387 FILE 100kb_dummy_file 256 385 AAAAAAAAAAAAAAAAAAAAAAAAAAAAA..."
=> Atualizando relógio para 4388
=> Atualizando relógio para 4389
Encaminhando mensagem "127.0.0.1:8082 4389 DL 100kb_dummy_file 256 390" 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 4391 FILE 100kb_dummy_file 160 390 AAAAAAAAAAAAAAAAAAAAAAAAAAAAA..."
=> Atualizando relógio para 4392

Download do arquivo 100kb_dummy_file finalizado.
```

Depois disso, se repetiu o processo alterando o valor do chunk padrão de 256, para os valores 512 e 1 repetindo o processo como se vê a abaixo:

```
Escolha um comando:
[1] Listar peers
[2] Obter peers
[3] Listar arquivos locais
[4] Buscar arquivos
[5] Exibir estatísticas
[6] Alterar tamanho de chunk
[9] Sair

>6
Digite novo tamanho do chunk:
>512
```

A partir do experimento foi possível obter o seguinte resultado:

Tam. chunk	N peers	Tam. arquivo	N	Tempo [s]	Desvio
256	2	1000	5	0.02708	0.01461
256	2	10000	6	0.19541	0.02146
256	2	100000	4	1.74895	0.18534
512	2	100000	4	0.78892	0.07245
512	2	10000	5	0.09965	0.02818
512	2	1000	5	0.01325	0.01116
1	2	1000	6	3.65975	0.39445
1	2	10000	3	31.09872	6.01086
1	2	100000	2	323.54466	26.32496



Partindo agora para a análise dos dados gerados, é possível observar uma influência significativa do tamanho do chunk no tempo total de download, especialmente à medida que o tamanho do arquivo aumenta.

Nos experimentos com arquivos de 1000 bytes, o tempo de download foi bastante baixo em todos os casos, mas ainda assim é perceptível que chunk sizes menores (como 1 byte) apresentaram um tempo substancialmente maior (3.66s) em comparação aos tamanhos de 256 (0.027s) e 512 bytes (0.013s). Isso indica que, mesmo para arquivos pequenos, há um custo associado ao overhead de coordenar múltiplas requisições.

Esse efeito se torna muito mais acentuado com arquivos maiores. Para arquivos de 10000 bytes, o tempo de download com chunk de 1 byte foi 31 segundos, enquanto com chunks de 256 e 512 bytes o tempo caiu para menos de 0.2 e 0.1 segundos, respectivamente. A diferença é ainda mais drástica para o arquivo de 100000 bytes, onde o tempo foi de 323.5 segundos com chunk de 1 byte, comparado a 1.74s com chunk de 256 bytes e 0.78s com chunk de 512 bytes.

Esses resultados demonstram que chunks muito pequenos degradam significativamente o desempenho, e isso está diretamente relacionado ao funcionamento da implementação.

Como o download é feito por meio de várias threads, cada uma requisitando um chunk específico para um peer, chunk sizes menores aumentam o número total de chunks, e conseqüentemente:

- Mais mensagens de download são enviadas e esperadas (uma por chunk);
- Maior esforço de ordenação ao final do processo (`file_results.sort()`), já que o número de entradas cresce consideravelmente, com complexidade  $O(n \log(n))$ ;
- Não há maior paralelismo, pois o número de threads é o mesmo, só aumentando o tamanho da requisição.

Resumindo, a análise dos dados mostra que o tamanho do chunk tem uma alta influência no desempenho da aplicação. Chunks pequenos podem saturar a infraestrutura de comunicação, travar sincronizações e aumentar consideravelmente o tempo de download, enquanto chunks maiores equilibram melhor o custo-benefício entre paralelismo e overhead.

### 3.2. Experimento sobre a influência do número de peers

Para esse segundo experimento foram inicializados sete peers, e inicialmente somente dois (8080, 8081) possuíam os arquivos.

```
Lista de peers:
[0] voltar para o menu anterior
[1] 127.0.0.1:8080 ONLINE
[2] 127.0.0.1:8081 ONLINE
[3] 127.0.0.1:8082 ONLINE
[4] 127.0.0.1:8083 ONLINE
[5] 127.0.0.1:8084 ONLINE
[6] 127.0.0.1:8086 ONLINE
```

Serão feitos 4 downloads de cada arquivo, e após isso os três arquivos serão adicionados em mais um peer e o processo será repetido, dessa forma analisando como o tempo é afetado pela quantidade de peers. Todo o experimento será realizado com chunk igual a 256 bytes.

	Nome	Tamanho	Peer
[ 0]	<Cancelar>		
[ 1]	100kb_dummy_file	100000	127.0.0.1:8080, 127.0.0.1:8081, 127.0.0.1:8082, 127.0.0.1:8083
[ 2]	10kb_dummy_file	10000	127.0.0.1:8080, 127.0.0.1:8081, 127.0.0.1:8082, 127.0.0.1:8083
[ 3]	1kb_dummy_file	1000	127.0.0.1:8080, 127.0.0.1:8081, 127.0.0.1:8082, 127.0.0.1:8083

Com isso foi possível obter as seguintes estatísticas:

Tam. chunk	N peers	Tam. arquivo	N	Tempo [s]	Desvio
256	2	100000	16	1.40852	0.11774
256	2	10000	16	0.15995	0.03423
256	2	1000	17	0.01573	0.00954
256	3	100000	24	0.86601	0.08920
256	3	10000	24	0.09338	0.02477
256	3	1000	18	0.01123	0.00787
256	4	100000	18	0.61555	0.07113
256	4	10000	14	0.07316	0.02069
256	4	1000	15	0.01097	0.00518
256	5	100000	26	0.61144	0.96862
256	5	10000	20	0.05753	0.01443
256	5	1000	20	0.01072	0.00580
256	6	100000	29	0.32136	0.02701
256	6	10000	20	0.04440	0.00958
256	6	1000	17	0.00917	0.00464

Com isso, podemos ver que o aumento no número de peers disponíveis traz ganhos significativos de desempenho, especialmente em arquivos de tamanho maior, evidenciando os benefícios do paralelismo na implementação.

Para arquivos de 100000 bytes, o tempo caiu de 1.40s com 2 peers para 0.32s com 5 peers e 0.19s com 6 peers. Isso ocorre porque mais peers permitem mais

threads em execução simultânea, cada uma baixando chunks em paralelo, como discutido no experimento anterior sobre o impacto dos chunks.

Esse ganho também aparece em arquivos de 10000 bytes, onde o tempo foi de 0.15s (2 peers) para 0.044s (6 peers). Já para arquivos pequenos como os de 1000 bytes, a diferença é menor: de 0.015s para 0.009s, pois o tempo total já é baixo e há menos chunks a distribuir.

Isso ocorre, pois pela implementação mais peers significam:

- Menor tempo de espera por chunk, pois há mais fontes respondendo;
- Menos contenção no lock, pois cada thread avança rapidamente;
- Mais mensagens paralelas, mas com menor peso individual.

Em resumo, os dados confirmam que o aumento no número de peers melhora substancialmente a performance, principalmente em arquivos maiores, tornando o sistema mais eficiente e escalável ao distribuir melhor a carga entre os participantes da rede.