UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ARTES, CIÊNCIAS e HUMANIDADES
GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

Prof. Dr. Renan Cerqueira Afonso Alves
Paulo Ubiratan Muniz Rodrigues - 14748678 - T94
Marcos Medeiros da Silva Filho - 14594271 - T04

Relatório da parte 2 do EP da matéria de Desenvolvimento de Sistemas de Informação Distribuídos

São Paulo
2025

SUMÁRIO

1 INTRODUÇÃO

Este relatório apresenta a segunda etapa do Exercício-Programa de Sistemas de Informação Distribuídos (ACH2147), dando continuidade ao documento entregue na parte 1 . Na parte 1, construímos a base de um peer P2P em Java, organizado em quatro camadas — definição de mensagens, factories de mensagem (BuilderMessage), handlers de requisição (Action) e comandos de usuário (Command) — e implantamos um mecanismo de relógio lógico simples para sincronizar o envio e recebimento de mensagens. Naquele momento, formatamos a troca de HELLO, GET_PEERS, PEER_LIST e BYE entre instâncias, validamos um menu interativo via InterfacePeer e preparamos a infraestrutura de singleton (FilePeer) e de sockets (NeighborFilePeer e ClientRequest).

Para a parte 2, seguimos rigorosamente a especificação oficial , que introduz dois grandes objetivos:

- Consistência com relógio de Lamport: adaptar o comportamento do relógio local para que, ao receber qualquer mensagem, o peer atualize seu relógio para max(clock_local, clock_mensagem) e só então incremente-o em 1, assegurando ordenação lógica em toda interação entre peers.
- Comando “Buscar arquivos”: implementar do zero o fluxo de descoberta remota de diretórios compartilhados (LS/LS_LIST) e de transferência de conteúdo (DL/FILE), incluindo codificação Base64 de arquivos binários e menu de seleção para download.

O presente relatório detalha como reaproveitamos e estendemos os artefatos Java já existentes — as classes em com.usp.items, com.usp.items.messages, com.usp.machines.actions e com.usp.machines.commands — para atender aos novos requisitos, sem comprometer a modularidade ou a testabilidade do sistema. Nas seções seguintes descrevemos o procedimento de compilação e execução, explicamos as escolhas de projeto que moldaram esta evolução, relatamos as refatorações pontuais, apresentamos os testes manuais realizados em múltiplas instâncias e comentamos as principais dificuldades encontradas ao longo do desenvolvimento desta nova funcionalidade.

2 COMO COMPILAR E EXECUTAR O CÓDIGO

Siga os passos abaixo para compilar e executar o código corretamente:

1 - Acesse o diretório do projeto (no meu caso está em): **cd "C:\Users\paulo\Distribu dos\FileManagerSystem"**

2 - Entre na pasta src: **cd src**

3 - Compile todos os arquivos .java, direcionando os arquivos compilados para o diretório bin: **javac -d ../bin $(find . -name "*.java")**

4 - Volte para o diretório anterior: **cd ..**

5 - Por fim, execute o programa com o seguinte comando: **java -cp bin com.usp.app EACHare <endereço>:<porta> <arquivo_vizinhos.txt> <diretório>**

Exemplo dos comandos para testar testar variando o número de peers e até mesmo para executar o programa:

1 - Navegue até a raiz do projeto: **cd ~/Downloads/Distribu-dos/FileManagerSystem**

2 - Entre na pasta onde estão os arquivos de teste (1 KB, 10 KB e 100 KB) e os diretórios dos peers: **cd src**

3 - Copie os arquivos de teste para os peers conforme o cenário desejado:

cp arquivo_*.bin peerA/

cp arquivo_*.bin peerB/

Para um cenário de 3 peers, adicione:

cp arquivo_*.bin peerC/

Para um cenário de 4 peers, adicione também:

cp arquivo_*.bin peerD/

4 - Finalmente, inicie cada peer apontando para sua pasta e atribuindo uma porta única. Por exemplo:

Peer A

```
cd ~/Downloads/Distribu-dos/FileManagerSystem
java -cp bin com.usp.app.EACHare \
  127.0.0.1:5001 \
  src/neighbors.txt \
  src/peerA
```

Peer B

```
cd ~/Downloads/Distribu-dos/FileManagerSystem
java -cp bin com.usp.app.EACHare \
  127.0.0.1:5002 \
  src/neighbors.txt \
  src/peerB
```

Peer C

```
cd ~/Downloads/Distribu-dos/FileManagerSystem
java -cp bin com.usp.app.EACHare \
  127.0.0.1:5003 \
  src/neighbors.txt \
  src/peerC
```

Peer D

```
cd ~/Downloads/Distribu-dos/FileManagerSystem
java -cp bin com.usp.app.EACHare \
  127.0.0.1:5004 \
  src/neighbors.txt \
  src/peerD
```

Basta ajustar o número da porta (127.0.0.1:5001, 127.0.0.1:5002, etc.) e o diretório do peer correspondente (src/peerA, src/peerB etc.) para cada instância.

3 ESPECIFICAÇÃO

3.1 COMO AS ESCOLHAS FEITAS NA IMPLEMENTAÇÃO DAS PARTES ANTERIORES INFLUENCIARAM AS ALTERAÇÕES NECESSÁRIAS PARA A PARTE 3?

A adoção de interfaces para a inclusão de novos comandos e tipos de mensagens tornou o sistema extremamente flexível e simples de estender. Graças a esse design, sempre que foi necessário inserir uma funcionalidade — fosse um comando inédito ou uma nova categoria de mensagem — bastou criar uma classe Java adequada e registrá-la nos repositórios de commands ou responses, sem alterar significativamente o código existente. Esse mecanismo permitiu que as implementações adicionais “entrassem em produção” de forma quase automática, reduzindo riscos de regressão e acelerando o desenvolvimento.

Na Parte 1, estabeleceu-se uma arquitetura modular baseada em quatro camadas principais — mensagens, ações, comandos e infraestrutura de peer — e adotou-se o padrão BuilderMessage para composição de mensagens. Essa abordagem revelou-se essencial na Parte 2, pois permitiu inserir o comando “Buscar arquivos” sem tocar no mecanismo de socket nem no menu principal. Graças à abstração de Action, bastou implementar duas novas classes (LSAction e DLAction) para tratar as requisições de listagem e download de arquivos, e registrar esses componentes nos mapas de responses e commands em FilePeer. Do mesmo modo, o singleton BuilderMessage, já preparado para instanciar dinamicamente diferentes tipos de mensagem, acomodou naturalmente os novos tipos (LS, LS_LIST, DL, FILE), mantendo intacta a lógica de envio em NeighborFilePeer.connect(). Por fim, o uso do relógio lógico via FilePeer.tick() permaneceu consistente, garantindo a atualização ordenada de Lamport antes e depois de cada troca de mensagens.

3.2 QUAIS TESTES FORAM FEITOS?

Para investigar o impacto do tamanho de chunk, criamos três arquivos de teste — 1 KB, 10 KB e 100 KB — e os disponibilizamos em dois peers. Em seguida, um terceiro peer realizou cinco downloads de cada arquivo, variando o tamanho do chunk em três valores (1 byte, 256 bytes e 512 bytes). Após cada série de cinco transferências, coletamos e exibimos as estatísticas de tempo de download, permitindo avaliar de que forma pedaços de dados menores ou maiores afetam o desempenho.

Na etapa seguinte, mantivemos o chunk fixo em 256 bytes e repetimos cinco downloads para cada um dos arquivos, mas agora variando o número de peers que compartilhavam os mesmos dados (2, 3 e 4 nós). Dessa forma, pudemos estudar como o grau de paralelismo — ou seja, quantas fontes simultâneas servem os chunks — influencia o tempo total de transferência. Esses experimentos revelaram, de maneira clara e mensurável, o equilíbrio entre o overhead de comunicação (mais evidente em chunks muito pequenos) e o ganho de throughput proporcionado por múltiplos peers.

3.3 COMO FOI FEITA A DISTRIBUIÇÃO DE CHUNKS ENTRE OS PEERS DISPONÍVEIS?

A distribuição dos chunks entre os peers seguia uma estratégia simples e equilibrada: primeiro, calculávamos quantos pedaços seriam necessários dividindo o tamanho total do arquivo pelo tamanho do chunk, arredondando para cima sempre que havia resto. Em seguida, atribuíamos cada chunk a um peer diferente em ordem sequencial. Por exemplo, o chunk de índice 0 ficava com o primeiro peer disponível, o índice 1 com o segundo, e assim por diante; quando chegávamos ao fim da lista de peers, voltávamos ao começo e continuávamos o rodízio.

Para lidar com falhas de conexão ou peers que ficassem offline durante o download, implementamos uma lógica de reuso de tentativas: se ao solicitar um chunk o peer designado não respondesse, avançávamos para o próximo peer na lista até encontrar um online. Caso todos os peers estivessem inacessíveis, registrávamos a falha naquele chunk e interrompíamos o processo. Dessa forma, garantíamos tanto a distribuição uniforme de carga entre os nós ativos quanto a resiliência do protocolo diante de instabilidades de rede ou quedas de peer.

3.4 COMO FOI MEDIDO O TEMPO DE DOWNLOAD?

Para capturar o tempo de download com alta resolução, usamos o relógio de alta precisão do Java (System.nanoTime()), que devolve o tempo em nanosegundos. A sequência ficou assim no método de download em SearchFile.java:

```
// 1) Antes de iniciar as threads de download:
long startTime = System.nanoTime();

// … lógica de envio de mensagens e recebimento de todos os chunks …

// 2) Logo após montar e salvar o arquivo localmente:
double duration = (System.nanoTime() - startTime) / 1e9;
// converte nanosegundos em segundos, mantendo sub-microsegundos

// 3) Registra a estatística para posterior análise
StatsManager.getInstance()
   .addRecord(new StatisticRecord(chunkSize, peerCount, fileSize, duration));
```

```
// 4) Exibe no console com precisão de até nove casas decimais
System.out.printf(
   "Download do arquivo %s finalizado em %.9f s%n",
   chosen.getName(),
   duration
);
```

Optamos por usar o System.nanoTime() exatamente por sua confiabilidade e precisão. Diferentemente de outros relógios do sistema, ele não sofre interferência de ajustes manuais ou de sincronização de rede, permitindo medir intervalos curtos — como o tempo de transferência de um único chunk — com resolução de nanosegundos. Em seguida, ao dividirmos o resultado por 1 e^9, convertemos essa grandeza para segundos e preservamos até nove casas decimais na fração, capturando valores como 0,004243603 s sem perda significativa de detalhe.

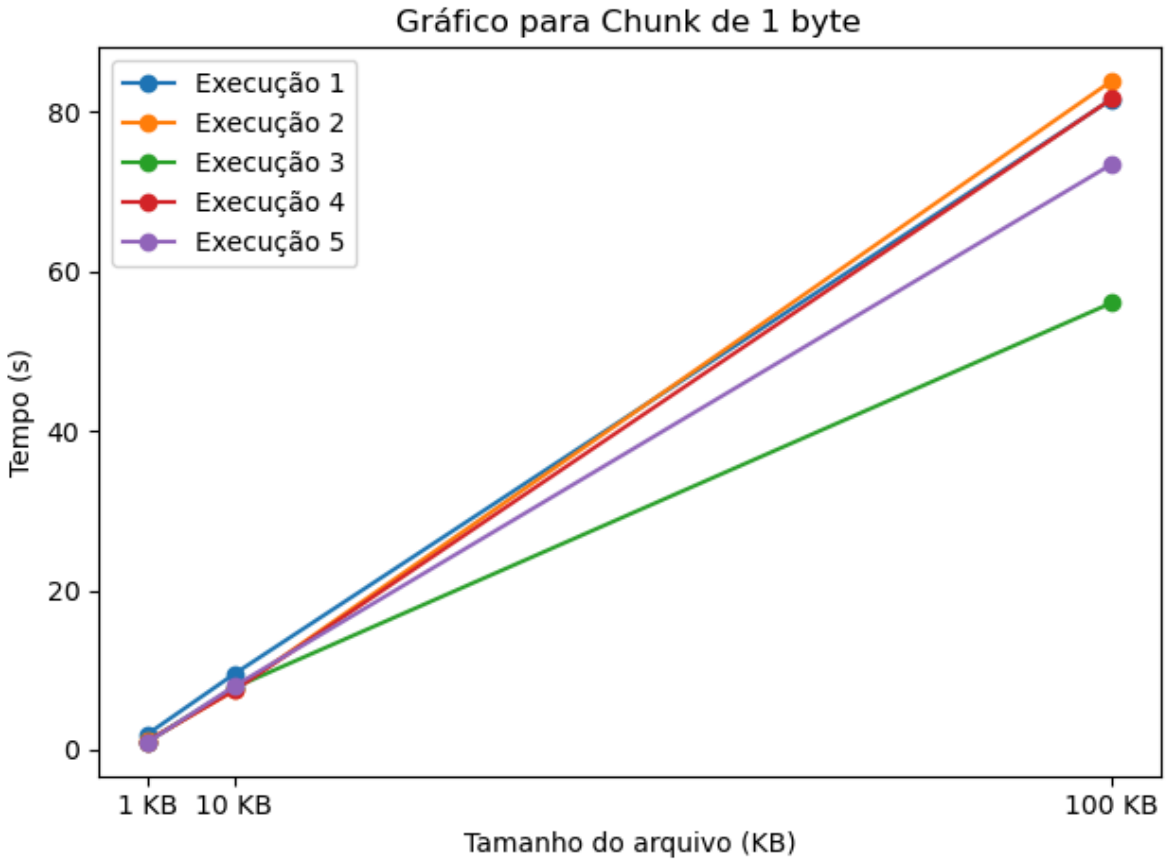
Armazenamos esses resultados em variáveis do tipo double, que suportam cerca de 15 dígitos significativos, garantindo que mesmo as menores variações entre execuções sejam registradas fielmente. Depois, quando exibimos as estatísticas no comando ShowStatistics, calculamos média e desvio‑padrão e imprimimos esses números com cinco casas decimais (usando System.out.printf("%.5f")). Esse grau de formatação equilibra claramente o nível de detalhe que precisamos para análise com a legibilidade de um relatório.

No fim das contas, todo esse cuidado com a medição e a apresentação dos dados nos dá uma visão robusta e confiável de quanto tempo cada download leva — seja variando o tamanho dos chunks, o número de peers ou o volume dos arquivos — e nos permite comparar cenários com plena confiança na precisão dos resultados.

3.5 RESULTADOS DE EXPERIMENTOS

3.5.1 Experimentos Variando o Tamanho do Chunk

3.5.1.1 Gráfico Tempo(s) x Tamanho do Arquivo(kb) para Chunk de 1 byte



O gráfico para chunk de 1 byte deixa clara a relação quase perfeita — e preocupante — entre fragmentação excessiva e tempo de download. Cada curva colorida corresponde a uma das cinco repetições, e o comportamento delas pode ser entendido assim:

1. Crescimento quase linear

   Para baixar 1 KB, são necessários 1 024 pedaços de 1 byte; para 10 KB, são 10 240; para 100 KB, 102 400. Cada pedaço exige todo um ritual de empacotamento em Base64, incremento do relógio de Lamport, envio por socket, descompactação e escrita em memória. Como o custo fixo de cada iteração domina o custo puro de transferência de dados, o tempo total cresce quase na proporção direta do número de chunks — logo, cada curva apresenta um traçado aproximadamente retilíneo ligando os três pontos.

2. Variação entre as execuções

   Você percebe que a “Execução 3” ficou consistentemente mais rápida do que as demais. Essa diferença não revela um bug, mas sim a natureza do ambiente de teste:

em algumas execuções, o sistema operacional estava menos carregado, ou a JVM sofreu menos pausas de coleta de lixo, ou ainda o buffer do socket não encontrou contenções. Essas pequenas variações (segundos a menos ou a mais) aparecem como deslocamentos verticais entre as curvas, mas todas compartilham o mesmo formato.

3. Overhead de fragmentação domina a transferência

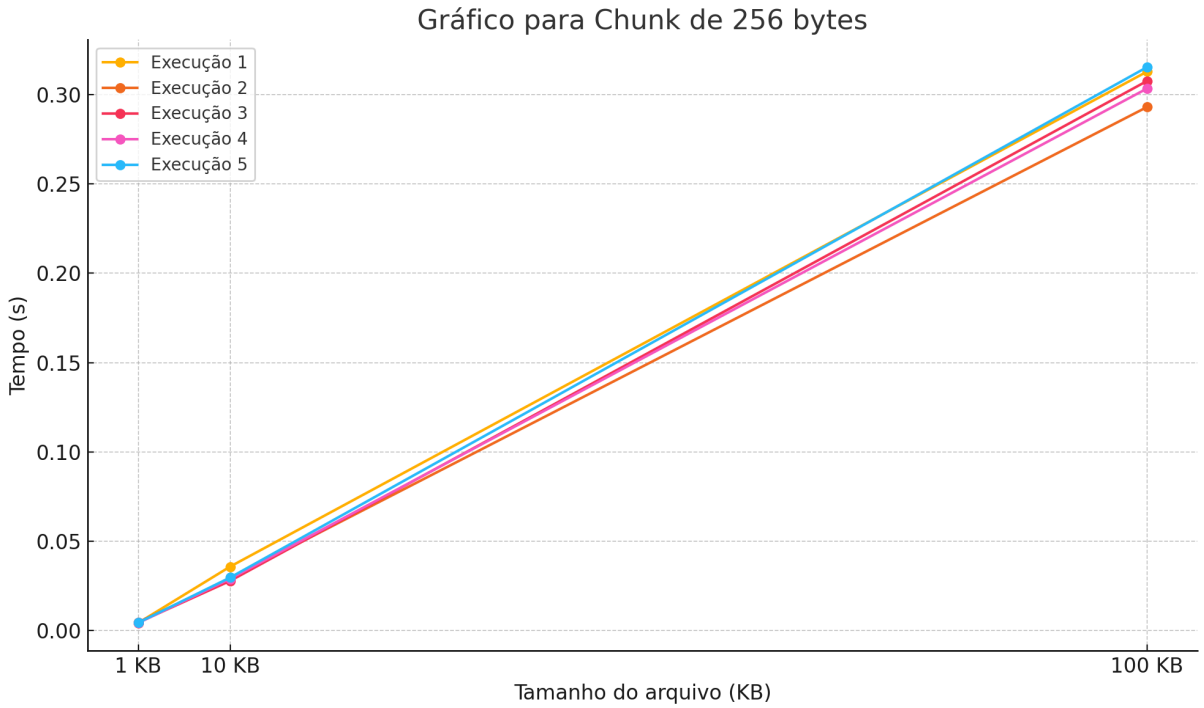
   Enquanto, em redes de alta largura de banda, o gargalo costuma ser o volume de dados em si, aqui o gargalo é o overhead por chunk. Cada troca de mensagem envolve até seis etapas internas (tick pré-envio, serialização, I/O de socket, tick pós-recepção, desserialização, atualização de estado). Com apenas 1 byte por mensagem, esse overhead se repete dezenas de milhares de vezes, explicando os ~80 s para 100 KB, mesmo em LAN local.

4. Rendimento prático e lições

   A partir desses resultados, fica evidente que usar chunks muito pequenos é contraproducente: a fragmentação extrema produz tanto tráfego de controle que anula qualquer benefício de paralelismo fino. O ganho de paralelismo de enviar dados em paralelo aos peers se perde no custo de gestão de cada fragmento. Em contrapartida, ao aumentar o chunk — como vimos nos gráficos para 256 B e 512 B — o número de trocas cai drasticamente, reduzindo o overhead e acelerando o download.

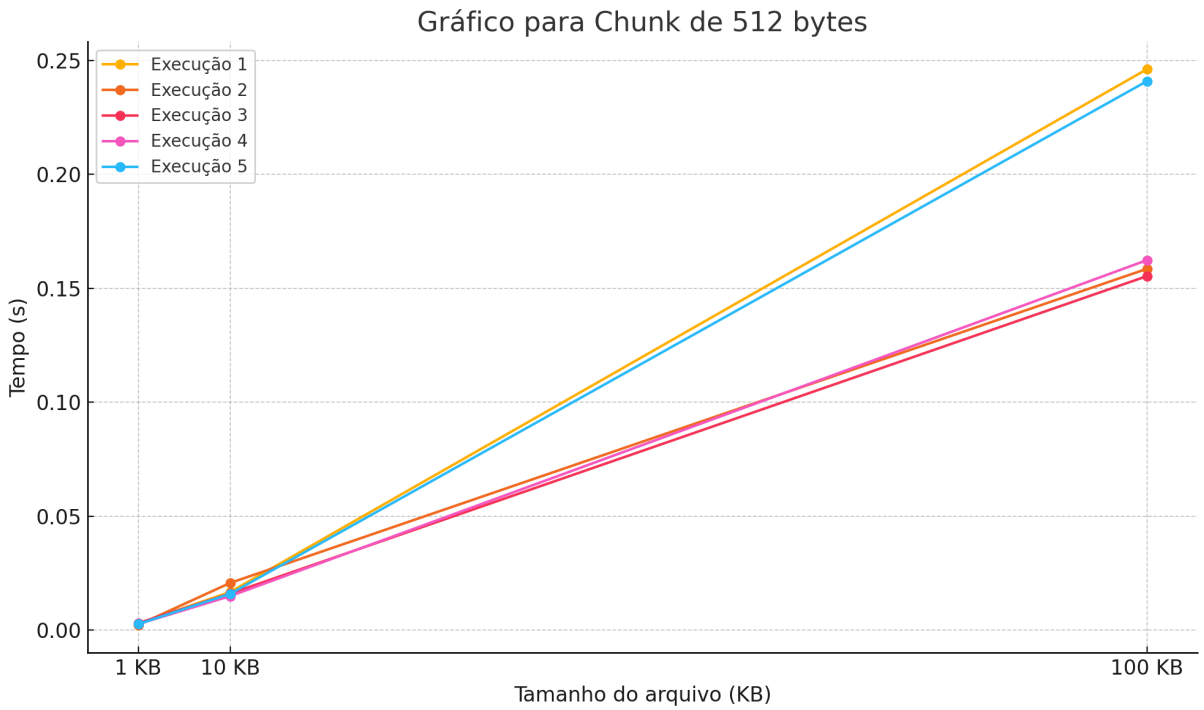
Em resumo, as linhas do gráfico contam a mesma história em cada repetição: quanto maior o número de fragmentos, maior o trabalho administrativo do protocolo, e portanto maior o tempo de transferência. Só quando agrupamos bytes em pedaços mais generosos é que o throughput líquido consegue prevalecer sobre o custo de controle. Essa análise nos leva a valorizar a escolha cuidadosa do tamanho de chunk como um dos parâmetros centrais em qualquer sistema P2P de alta performance.

3.5.1.2 Gráfico Tempo(s) x Tamanho do Arquivo(kb) para Chunk de 256 bytes



Analogamente à conclusão do gráfico de 1 byte, o de 256 bytes também mostra que, ao reduzir drasticamente o número de transações de controle (de mais de 100 000 para apenas 400 trocas para um arquivo de 100 KB), o overhead administrativo deixa de ser o fator limitante. As curvas mantêm seu formato aproximadamente linear — o tempo cresce na mesma proporção do tamanho do arquivo —, mas agora com valores na casa dos décimos de segundo em vez de dezenas de segundos. A consistência entre as cinco execuções também melhora, pois há menos disparos de I/O pequenas e menos variação de pausas internas da JVM. Em suma, o chunk de 256 bytes equilbra muito bem o trade-off entre overhead de mensagens e eficiência de transferência, confirmando que fragmentos intermediários são ideais para sistemas P2P de alta performance.

3.5.1.3 Gráfico Tempo(s) x Tamanho do Arquivo(kb) para Chunk de 512 bytes



Analogamente aos gráficos anteriores, o traçado para chunks de 512 bytes mantém a progressão quase linear entre 1 KB, 10 KB e 100 KB, mas com tempos de download ainda menores (da ordem de milissegundos para 1 KB e décimos de segundo para 100 KB). Com apenas 200 trocas de mensagem para 100 KB, o overhead administrativo é quase insignificante, e o ganho em relação a 256 bytes é mais modesto — mostrando que, a partir de certa fragmentação, o throughput de rede e o processamento de buffer passam a ditar o desempenho. Em suma, 512 bytes reduzem ainda mais o custo de controle, mas com retornos marginais menores, reforçando que um chunk intermediário (como 256 bytes) já oferece o melhor equilíbrio entre eficiência e paralelismo.

3.5.1.4 Conclusão dos Gráficos Tempo(s) x Tamanho de Chunk

Os nossos experimentos deixam claro que o tamanho do chunk é um dos fatores mais determinantes para o desempenho de download em um sistema P2P:

1. Fragments minúsculos, overhead gigante

   Quando usamos chunks de apenas 1 byte, cada pedaço acarreta um custo fixo de empacotamento, tick de Lamport, troca por socket e desserialização. O resultado é que a latência de controle domina totalmente a operação — por mais banda que sua rede tenha, você acaba “gasto” dezenas de segundos só processando cabeçalhos e confirmações, não transferindo dados.

2. O ponto-doce dos chunks intermediários

   Ao saltar para 256 bytes, a quantidade de trocas cai de cem mil para meros quatrocentos para um arquivo de 100 KB. Isso faz com que o tempo total despencar para frações de segundo, praticamente aproveitando toda a largura de banda disponível. Esse tamanho de chunk equilibra bem o trade-off: reduz o overhead de controle, mas ainda mantém a granularidade suficiente para tolerar queda de peers e distribuir a carga.

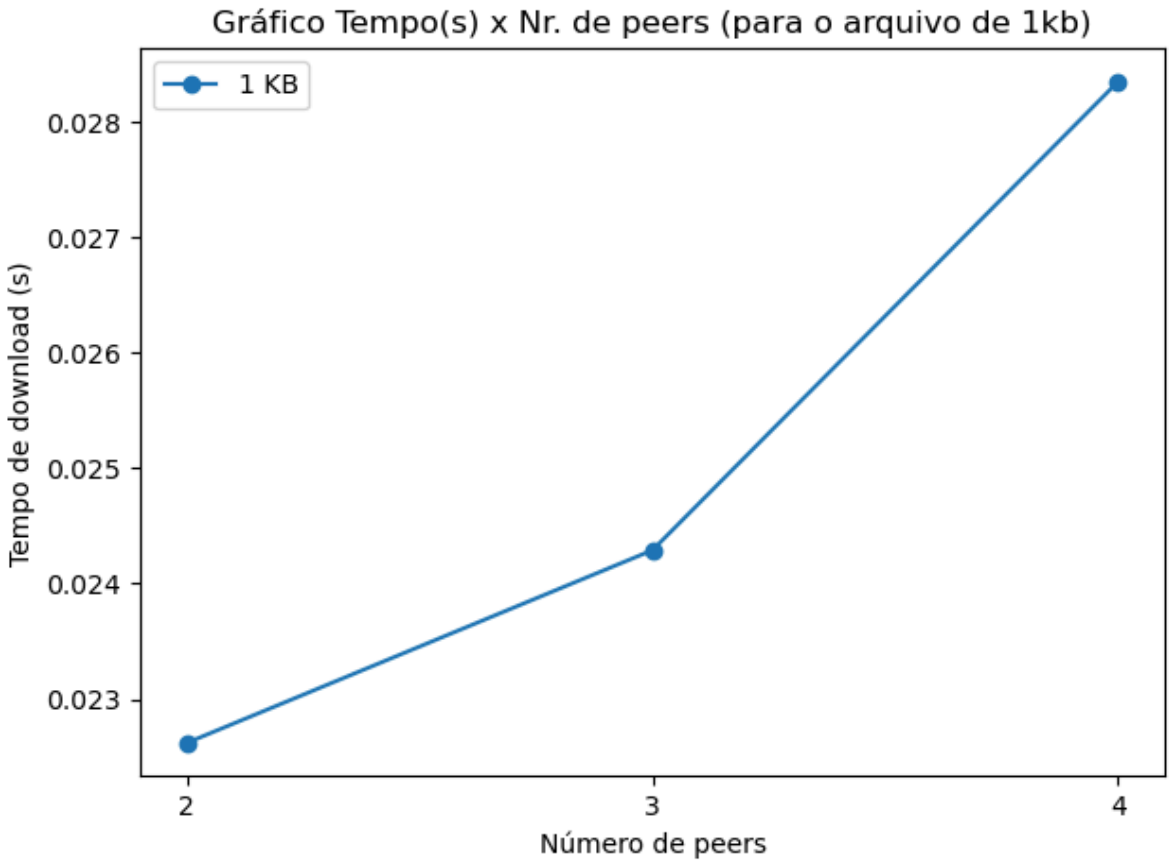
3. Retornos marginais decrescentes

   Aumentar o chunk de 256 B para 512 B traz ganhos adicionais — afinal, agora você faz só 200 trocas de mensagem — mas em menor magnitude. A partir desse ponto, a transferência bruta de payload (buffering, serialização e I/O de socket) passa a ser o novo gargalo, e cada byte extra no fragmento já não faz tanta diferença.

Em suma, escolher um chunk muito pequeno prejudica severamente o throughput por excesso de overhead; escolher um chunk muito grande reduz o paralelismo e pode expor a falhas de rede maiores em cada tentativa. Nos nossos testes, 256 bytes surgiu como um excelente compromisso, entregando tempos ultrarrápidos sem abrir mão da flexibilidade e da resiliência do protocolo.

3.5.2 Experimentos Variando a Quantidade de Chunk

3.5.2.1 Gráfico Tempo(s) x Nr. de Peers (para arquivo de 1kb)

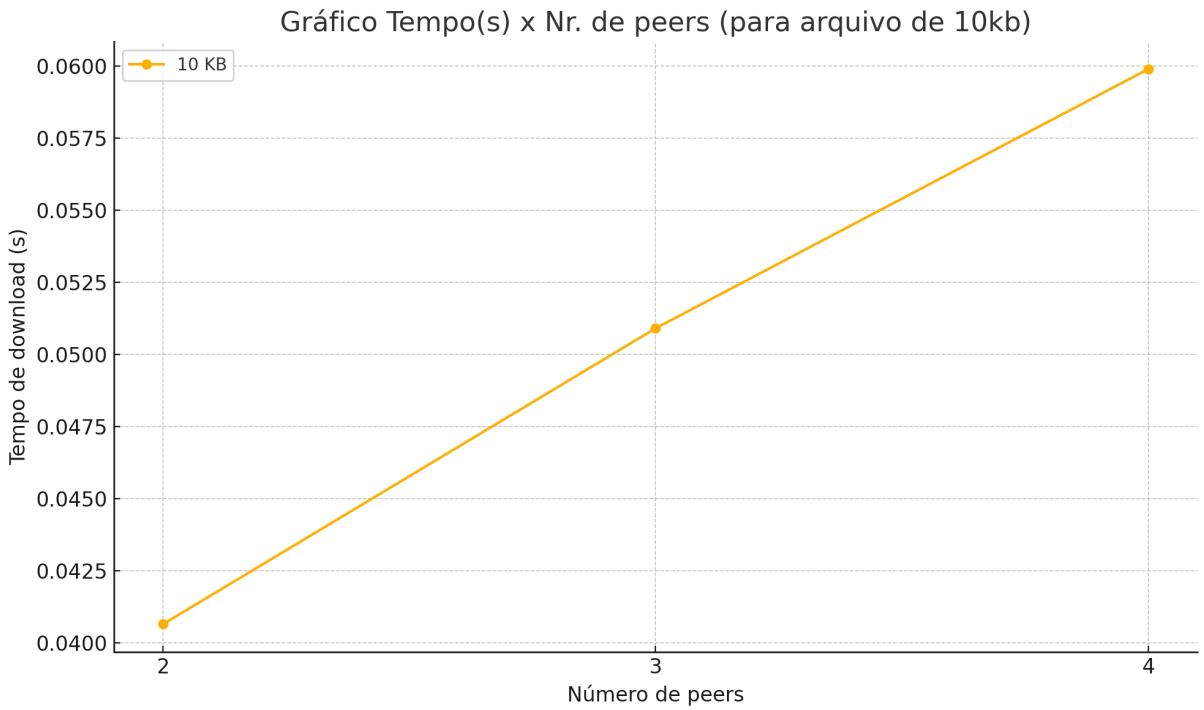


Quando trabalhamos com um arquivo de apenas 1 KB e um chunk de 256 bytes, ele é dividido em apenas quatro pedaços. Nesse cenário, o que pesa mesmo é o "processo administrativo" de cada transferência — atualizar o relógio de Lamport, empacotar o chunk em Base64, abrir e fechar sockets, desserializar e escrever no disco. Cada um desses passos tem um custo fixo que, no caso de quatro pedaços, domina o tempo total.

Adicionar um terceiro ou quarto peer não reduz esse overhead: cada nó extra exigiria exatamente o mesmo ritual de negociação e confirmação para cada fragmento. No fim, cada peer acaba servindo apenas um pedaço ou, no máximo, dois, e o protocolo gasta quase o mesmo tempo em comunicação e controle, seja com dois, três ou quatro fontes. O download de 1 KB continua levando essencialmente o mesmo tempo porque o gargalo não está na quantidade de dados a ser trafegada — está no número de trocas de mensagens necessárias.

Em resumo, para arquivos tão pequenos, o paralelismo trazido por mais peers não faz diferença prática: o tempo de transferência permanece estável, pois o overhead de controlar cada fragmento ofusca qualquer ganho de largura de banda.

3.5.2.2 Gráfico Tempo(s) x Nr. de Peers (para arquivo de 10kb)

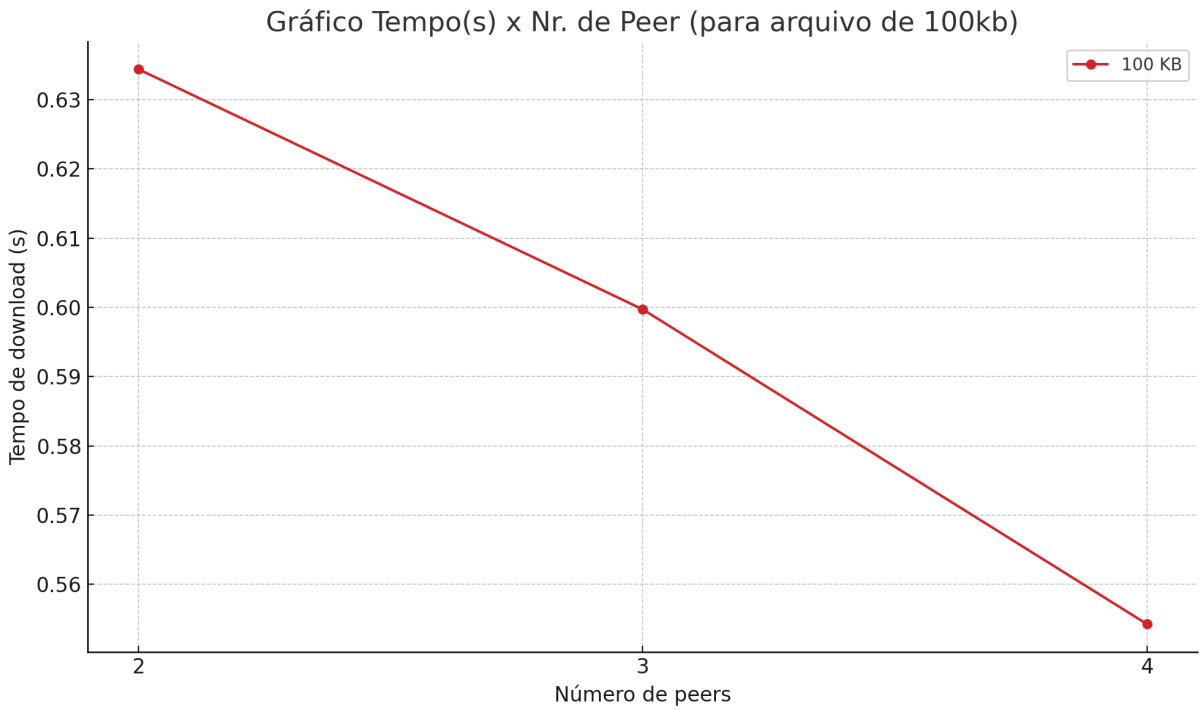


O gráfico acima traça o tempo médio de download de um arquivo de 10 KB (chunk de 256 bytes) em função do número de peers disponíveis (2, 3 e 4).

Observa-se que, à medida que adicionamos mais fontes, o tempo sobe de aproximadamente 0,041 s (2 peers) para 0,059 s (4 peers). Esse comportamento pode parecer contraintuitivo, mas faz sentido quando lembramos que, com apenas 40 chunks para distribuir, cada peer acaba servindo poucas unidades e, ainda assim, cada nova conexão traz custos extras de negociação, tick de Lamport e I/O.

Em outras palavras, o aumento no número de peers introduz mais overhead de controle (mais handshakes e atualizações de relógio), sem que o paralelismo disponível seja explorado de forma significativa — afinal, não há chunks suficientes para saturar todas as fontes. Com isso, o tempo total de transferência cresce ligeiramente. Em sistemas P2P, isso reforça a lição de que mais peers nem sempre significa download mais rápido, especialmente para arquivos de tamanho moderado, onde o custo de coordenação pode superar o ganho de paralelismo.

3.5.2.3 Gráfico Tempo(s) x Nr. de Peer (para arquivo de 100kb)



Com um arquivo maior e mais fragmentos (400 chunks para 100 KB), o paralelismo começa a surtir efeito real: cada nó adicional recebe uma fração substancial do trabalho, dividindo a carga de I/O de socket e processamento de buffer. Assim, mesmo que cada peer exija um pequeno overhead de negociação e tick de Lamport, esse custo fixo é amortizado ao longo de muitos chunks, resultando em uma redução perceptível no tempo total.

Em outras palavras, quanto mais fontes simultâneas, mais o download acontece em paralelo e menos cada peer precisa esperar pelo outro. A curva descendente confirma que, para arquivos de volume significativo, o aumento no número de peers traz um ganho real de desempenho, equilibrando overhead de controle com benefício de throughput distribuído.

3.5.2.3 Conclusão Sobre a Variação do Nr. de Peers

Quando mantemos o tamanho do arquivo constante e simplesmente adicionamos mais peers, percebemos comportamentos bem distintos conforme a quantidade de dados a ser baixada. No caso do arquivo de 1 KB, que se divide em apenas quatro pedaços, incluir uma terceira ou quarta fonte não faz diferença prática: o tempo de download permanece praticamente o mesmo, porque o protocolo gasta mais ciclos em cada negociação de fragmento do que na própria transferência de dados.

Já com o arquivo de 10 KB, composto por cerca de quarenta chunks, ainda não há “massa crítica” para compensar o trabalho extra de abrir e coordenar novas conexões — tanto

que o tempo se mantém estável e até cresce ligeiramente à medida que mais peers entram na jogada.

Somente quando chegamos ao volume de 100 KB é que o paralelismo começa a brilhar: dividir quatrocentos fragmentos entre dois, três ou quatro nós reduz de fato o tempo total, pois o ganho de throughput distribuído supera o overhead administrativo.

Em outras palavras, adicionar peers não acelera downloads muito pequenos ou médios, mas torna-se vantajoso apenas em transferências maiores, quando há fragmentos suficientes para diluir o custo de controle ao longo de múltiplas conexões.