Sistemas Distribuídos
Trabalho Prático 2
Transferência de Arquivos Peer-to-Peer

Letícia de Oliveira Soares
Mateus Gonçalves

Este relatório explica o funcionamento do arquivo run_test.sh e dos componentes principais do sistema P2P descrito. Ele também faz uma análise dos casos de teste pedidos no enunciado

Repositório: https://github.com/leticiaasoares/SD_TP2/tree/main

1. Arquivo run_test.sh

O script remove diretórios antigos de logs, blocos e vizinhos e cria novos vazios. Isso garante um ambiente limpo para cada execução. O comando dd gera um arquivo binário com dados aleatórios. Esse arquivo será fragmentado pelo seeder e distribuído entre os peers.

O script chama seed_prepare dentro de utilities.py. Essa função: divide o arquivo em blocos do tamanho especificado, gera um arquivo meta.json contendo informações sobre cada bloco, armazena os blocos na pasta blocks/.

O peer1 age como seeder, pois ele inicializa com todos os blocos. Para cada peer, o script cria um arquivo JSON contendo a lista dos outros peers com seus IPs e portas. Assim, todos sabem com quem podem se comunicar.

O script executa p2p_peer.py para cada peer, passando:

- --peer-id
- --ip
- --port
- --neighbors
- --meta
- --storage

A saída de cada peer é redirecionada para logs/peerX.log. Ao final, o script compara o hash SHA256 do arquivo reconstruído por cada peer com o arquivo original.

2. Arquivo p2p_peer.py

O arquivo p2p_peer.py implementa o comportamento de um peer da rede P2P. Cada peer atua simultaneamente como servidor e cliente, permitindo:

- Receber pedidos de blocos

- Enviar blocos para outros peers
- Baixar blocos que faltam
- Reconstruir o arquivo final

2.1 Carregamento dos metadados:
O peer carrega o arquivo meta.json, que contém:
- O nome do arquivo original
- O número total de blocos
- O tamanho dos blocos
- O mapeamento de blocos

Isso permite ao peer saber quais blocos existem e quais ele precisa obter.

2.2 Carregamento dos blocos locais:
O peer verifica quais blocos já existem em sua pasta de armazenamento. Se for o seeder, ele possui todos os blocos. Se for um peer comum, normalmente inicia com zero blocos.

2.3 Carregamento da lista de vizinhos:
O peer lê o arquivo JSON contendo IP e porta de cada vizinho. Ele utiliza essa lista para tentar baixar
blocos.

2.4 Execução do servidor interno:
O peer inicia um pequeno servidor TCP que escuta na porta definida. Esse servidor recebe pedidos de blocos, verifica se possui o bloco solicitado, se possuir, envia o bloco ao peer solicitante, registra cada requisição no log.
Esse servidor permite que outros peers baixem blocos.

2.5 Mecanismo de download de blocos:
O peer inicia uma rotina de download, que percorre a lista de blocos faltantes, contata cada vizinho solicitando blocos, recebe blocos e armazena na pasta local, atualiza a lista de blocos faltantes, para quando todos os blocos forem obtidos
A comunicação é feita via sockets TCP.

2.6 Reconstrução do arquivo final:
Quando o peer obtém todos os blocos ele abre um novo arquivo de saida chamado peerX_RECONSTRUCTED_nomeDoArquivo, lê cada bloco em ordem numérica e reescreve o arquivo original em disco.

2.7 Registro em logs:
Durante toda a execução, o peer escreve informações como blocos recebidos, quem enviou blocos, erros de conexão e início e fim do processo de reconstrução.
Esses logs aparecem no diretório logs/peerX.log.

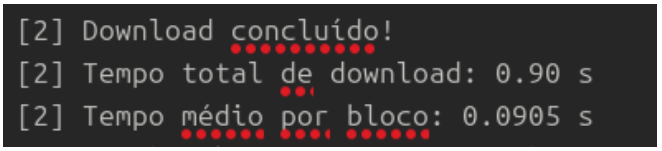
3. Testes

3.1 Quantidade de Peers

Ao aumentar a quantidade de peers de 2 para 4, o sistema tende a distribuir melhor as requisições, permitindo que vários peers enviem blocos simultaneamente. Isso geralmente resulta em redução do tempo total de download, maior paralelismo e melhor balanceamento de carga.

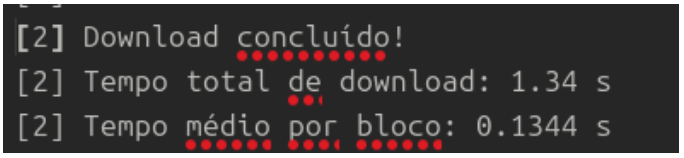
Com poucos peers, a transmissão se concentra em poucas fontes, aumentando o tempo de espera entre blocos.

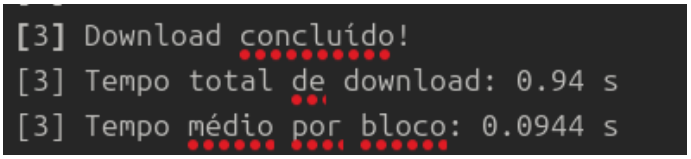
2 peers:

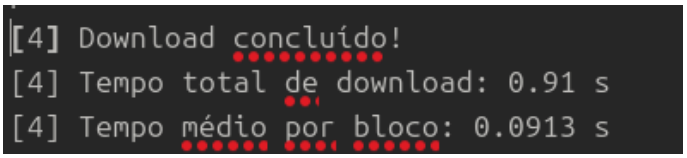


Nesse cenário, o peer possui apenas uma ou poucas fontes para solicitar blocos. Isso reduz o paralelismo: muitos blocos precisam esperar o peer vizinho ficar livre para enviar dados. O gargalo de transmissão fica mais evidente porque a carga não é distribuída entre vários emissores.

4 peers:







Com mais peers disponíveis, cada bloco pode ser obtido de múltiplas fontes. Isso aumenta o paralelismo das transferências, reduz o tempo ocioso e distribui a carga de envio entre vários vizinhos. O tempo médio por bloco cai drasticamente, de aproximadamente 0.9 s por bloco (2 peers) para cerca de 0.09 a 0.13 s por bloco (4 peers).

Aumentar o número de peers melhora significativamente o desempenho do sistema, reduzindo o tempo médio por bloco e permitindo maior paralelismo. A rede fica menos dependente de um único peer, resultando em uma distribuição mais equilibrada das transferências e, portanto, maior velocidade global.

3.2 Tamanho do Bloco

Nos testes com blocos grandes (4090 bytes) e blocos pequenos (1024 bytes), todos os peers reconstruíram corretamente o arquivo, e os checksums SHA-256

confirmaram a integridade dos dados em ambos os casos. A remontagem ocorreu sem falhas, mostrando que o protocolo funciona bem tanto com poucos blocos quanto com muitos.

Os logs mostraram que os blocos vieram das fontes esperadas. Apesar de algumas mensagens isoladas de “Failed to respond”, todas foram seguidas por respostas bem-sucedidas, indicando estabilidade no processo, mesmo quando o número de blocos aumenta.

Em relação ao desempenho, blocos maiores resultaram em menos pedidos, porém maior tempo médio por bloco (0.27–0.41s). Já blocos menores aumentaram o número de blocos, porém reduziram o tempo médio por bloco (0.09–0.13s). Assim, blocos pequenos tornam a transferência mais granular e eficiente por unidade de bloco, mas exigem mais trocas entre peers.

No geral, ambos os tamanhos funcionaram corretamente: blocos grandes ofereceram simplicidade (menos fragmentos), enquanto blocos pequenos ofereceram maior fluidez nas requisições. O protocolo mostrou robustez para ambos os cenários.

3.3 Arquivo Pequeno (File A)

Nos arquivos pequenos, tanto 10 KB quanto 20 KB foram transferidos rapidamente e sem erros. Em ambos os casos, todos os blocos foram recebidos e os checksums confirmaram a integridade do arquivo.

Para 10 KB (10 blocos), o peer 2 levou 0.90 s (0.0904 s por bloco) e o peer 3 levou 0.50 s (0.0503 s por bloco). Já no arquivo de 20 KB (20 blocos), o peer 2 completou em 0.60 s (0.0302 s por bloco), enquanto o peer 3 demorou 1.01 s (0.0504 s por bloco). Assim, mesmo com o dobro de blocos, o desempenho se manteve estável, e em alguns casos até melhor, mostrando que o protocolo lida bem com transferências pequenas e rápidas.

No geral, arquivos pequenos, seja com 10 ou 20 blocos, foram transmitidos rapidamente, com baixa latência e sem impacto perceptível na integridade ou estabilidade do protocolo. O sistema se mostrou eficiente para esse tipo de cenário de transferência leve.

3.4 Arquivo Médio (File B)

Nos testes com arquivos médios, o sistema fragmentou corretamente o FileB.bin em uma quantidade significativa de blocos: 1024 blocos para o arquivo de 1 MB e 5120 blocos para o arquivo de 5 MB, ambos usando blocos de 1024 bytes. Esses valores confirmam que o protocolo P2P lida com fragmentação extensa, já que os peers precisam gerenciar milhares de blocos durante a transmissão.

O tempo total de download aumentou proporcionalmente ao tamanho do arquivo, como esperado: de 13 segundos para 1 MB para cerca de 61 segundos para 5 MB. Isso confirma que o gargalo não está na fragmentação, mas simplesmente no volume total de dados a serem transmitidos.

Os logs também mostram que, nos arquivos maiores, há mais mensagens RESP_BLOCK, o que é natural, mas não aparecem erros relevantes. Todas as

respostas de blocos indicam sucesso, mesmo com milhares de requisições. O comportamento ocasional de peers com "REQ_HAVE - blocks set()" no peer 3 é normal no início da sessão e não impactou o download.

No geral, os resultados mostram que o protocolo lida corretamente com fragmentação extensa, mantém integridade dos dados, escala bem para milhares de blocos e distribui o arquivo sem inconsistências, mesmo com um número maior de peers e requisições.

3.5 Arquivo Grande (File C)

Nos testes com arquivos grandes (10 MB e 20 MB), o sistema mostrou comportamento estável mesmo com milhares de blocos. Em ambos os tamanhos, todos os peers conseguiram baixar e remontar os arquivos sem erros, e os hashes SHA-256 sempre coincidiram com o original, indicando integridade total. O tempo médio por bloco permaneceu praticamente igual nos dois cenários, cerca de 0.012 segundos, mostrando que o aumento do tamanho do arquivo não afetou o desempenho por bloco. Como resultado, dobrar o tamanho do arquivo apenas dobrou o tempo total de download, o que indica um funcionamento linear e previsível. Os logs mostram grande quantidade de respostas RESP_BLOCK, mas sem falhas persistentes. No geral, o protocolo se manteve estável, escalável e confiável mesmo em transferências muito maiores, o que demonstra boa robustez para cenários de alto volume de dados.

3.6 Configuração de Vizinhos

A configuração de vizinhos tem impacto direto no comportamento e no desempenho do protocolo P2P durante os testes. Em um sistema P2P, cada peer depende de seus vizinhos para transmitir blocos, manter atualizações e detectar falhas. Portanto, a forma como os vizinhos são selecionados e quantos são mantidos altera a dinâmica da rede.

Se o número de vizinhos for muito alto, a rede fica mais estável e resiliente, mas isso aumenta o tráfego de mensagens de controle e pode gerar sobrecarga. O peer passa a manter muitas conexões ativas, o que exige mais processamento, mais verificações de estado e mais trocas de metadados.

Em termos de análise, a configuração de vizinhos influencia diretamente nos seguintes pontos: velocidade de propagação de blocos, estabilidade durante longas transferências, redundância de caminhos para recuperação de falhas e nível de sobrecarga da rede. Alterar o valor de vizinhos permite avaliar como o protocolo P2P se adapta a diferentes topologias e como o fluxo de dados responde ao aumento ou redução de conexões diretas.