# Relatório Técnico - Entrega 2

Implementação de um Sistema P2P com Múltiplas Conexões

**Grupo 3**

João Pedro Queiroz Rodrigues - 170121348

João Victor Alves dos Santos - 180033760

17 de junho de 2025

**Resumo**

Este relatório detalha a implementação e os testes da segunda fase do sistema P2P desenvolvido para a disciplina de Teleinformática e Redes 2. O foco desta entrega foi habilitar a transferência de arquivos entre peers, utilizando uma estratégia de paralelismo com múltiplas conexões TCP para otimizar a velocidade de download, e garantindo a integridade dos dados através de checksums SHA-256.

## Sumário

# 1 Estrutura dos Chunks e Validação

Para otimizar a transferência e validação de arquivos na rede P2P, foi adotada uma estratégia de divisão de arquivos em pedaços menores, chamados *chunks*.

## 1.1 Divisão em Chunks

Qualquer arquivo a ser compartilhado é primeiramente dividido em chunks de tamanho fixo. Em nossa implementação, definimos um `CHUNK_SIZE` de 1MB (1.024 x 1.024 bytes). O último chunk de um arquivo pode, naturalmente, ser menor que este valor. Essa abordagem permite que um peer comece a baixar um arquivo sem precisar esperar que outro peer o tenha por completo, além de ser a base para o download paralelo.

## 1.2 Validação de Integridade

A integridade dos dados é crucial e é garantida em dois níveis através do uso de hashes SHA-256:

1. **Validação por Chunk:** Antes de anunciar um arquivo, o peer "seeder"calcula o hash SHA-256 de cada um de seus chunks. Essa lista de hashes é enviada ao tracker junto com os metadados do arquivo. Quando um peer "leecher"baixa um chunk, ele imediatamente calcula o hash do dado recebido e o compara com o hash esperado (informado pelo tracker). Se os hashes não baterem, o chunk é descartado e pode ser solicitado novamente. Isso garante a integridade contra corrupção de dados durante a transferência. O trecho de código abaixo ilustra essa validação no peer downloader.

```
# ... dentro da thread de download ...
response = s.recv(CHUNK_SIZE + 1024)

# Valida o hash do chunk recebido
if hashlib.sha256(response).hexdigest() == expected_hash:
    # Hash correto, salva o chunk
    with open(chunk_path, 'wb') as f:
        f.write(response)
    success = True
else:
    # Falha na verificacao, o chunk sera baixado novamente
    log("Falha na verificacao de hash...", "WARNING")
```

Listing 1: Verificação do hash do chunk no DownloaderThread.

2. **Validação do Arquivo Final:** Após todos os chunks serem baixados e validados individualmente, eles são reunidos para reconstruir o arquivo original. Como uma verificação final, o sistema calcula o hash SHA-256 do arquivo recém-montado e o compara com o hash global do arquivo, que também foi fornecido pelo tracker. Isso assegura que a reconstrução ocorreu corretamente, sem chunks faltantes, duplicados ou em ordem errada.

# 2 Estratégia de Paralelismo

Para acelerar significativamente o processo de download, implementamos uma estratégia de paralelismo baseada em *multithreading*, utilizando o módulo `threading` do Python.

O paralelismo ocorre no lado do peer que está realizando o download (leecher). A lógica é a seguinte:

1. Ao iniciar um download, o peer cria uma fila de trabalho (instância de `queue.Queue`) contendo todos os chunks que precisam ser baixados.
2. O sistema então instancia e inicia um número pré-definido de threads trabalhadoras (`NUM_DOWNLOAD_THREADS`).
3. Cada uma dessas threads opera de forma independente: ela retira um chunk da fila, conecta-se a um peer que o possua, realiza o download e a validação.
4. Como as threads rodam concorrentemente, múltiplos chunks podem ser baixados ao mesmo tempo, cada um em sua própria conexão TCP. Isso utiliza melhor a banda disponível e reduz o tempo total de espera.

No lado do peer que envia os dados (seeder), o servidor TCP também utiliza threads para lidar com as múltiplas conexões simultâneas vindas do leecher, garantindo que uma requisição não bloqueie as outras.

# 3 Testes de Desempenho

Para validar empiricamente a eficácia da nossa estratégia de paralelismo, realizamos testes de download de um arquivo de **9.1 MB** (9.542.252 bytes), variando o número de threads de download. O objetivo é medir o tempo total para baixar o arquivo completo em cada cenário.

## 3.1 Metodologia

Os testes foram conduzidos em uma máquina local, com um peer atuando como seeder e outro como leecher. O tempo foi cronometrado desde o início da requisição de download até a validação bem-sucedida do arquivo final. A constante `NUM_DOWNLOAD_THREADS` no arquivo `peer_client.py` foi alterada para cada teste.

## 3.2 Resultados

Os resultados obtidos estão compilados na tabela abaixo.

| **Nº de Conexões Paralelas** | **Tempo de Download (segundos)** |
|---|---|
| 1 (Sequencial) | 3.85 |
| 2 | 2.12 |
| 4 | 1.23 |
| 8 | 1.09 |

Tabela 1: Tempo de download de um arquivo de 9.1 MB versus o número de conexões paralelas.

## 3.3 Análise

Os resultados apresentados na Tabela 1 demonstram claramente a eficácia da abordagem de download paralelo. Com uma única conexão (modo sequencial), o download levou 3.85 segundos. Ao dobrar o número de conexões para 2, o tempo foi reduzido em aproximadamente 45%, para 2.12 segundos.

O ganho de desempenho mais significativo ocorreu ao saltar para 4 conexões paralelas, nosso valor padrão, que registrou um tempo de 1.23 segundos — uma redução de quase 68% em relação ao caso sequencial. Isso indica que, para este cenário, o principal gargalo era a limitação da transferência de dados em uma única thread.

Ao aumentar o número de conexões para 8, observamos um ganho de desempenho marginal, com o tempo caindo apenas para 1.09 segundos. Essa diminuição no retorno sugere que o sistema começou a atingir outros gargalos, como o overhead de gerenciamento e escalonamento de um número maior de threads pelo sistema operacional, ou até mesmo os limites de I/O (escrita em disco) da máquina local. Conclui-se que um número entre 4 e 8 conexões paralelas representa o ponto ótimo de equilíbrio entre performance e uso de recursos para esta aplicação em nosso ambiente de teste.