Sistema P2P de Transferência de Arquivos

Trabalho Prático - Sistemas Distribuídos

Arthur Soares Luiz Carlos Jamil Apicela

14 de novembro de 2025

Repositório do Código-Fonte

URL: https://github.com/ArthurSHigino/SD_2

---

1. Introdução

Este trabalho apresenta a implementação de um sistema elementar de transferência de arquivos utilizando o modelo Peer-to-Peer (P2P) para a disciplina de Sistemas Distribuídos. O objetivo principal foi desenvolver uma solução onde múltiplos peers atuam simultaneamente como clientes e servidores, permitindo a distribuição eficiente de arquivos através de fragmentação em blocos.

Nossa implementação focou em criar um sistema funcional que demonstrasse os conceitos fundamentais de sistemas distribuídos P2P, incluindo fragmentação de dados, comunicação entre peers e verificação de integridade.

2. Decisões de Projeto

2.1 Escolha da Linguagem

Optamos por Python 3 como linguagem de implementação pelas seguintes razões:

- **Facilidade com Sockets**: Biblioteca `socket` nativa e bem documentada
- **Threading Simplificado**: Suporte robusto para concorrência com `threading`
- **Serialização JSON**: Facilita o protocolo de comunicação entre peers
- **Bibliotecas Criptográficas**: `hashlib` para verificação de integridade SHA-256
- **Prototipagem Rápida**: Permite foco na lógica P2P sem complexidade sintática

2.2 Arquitetura do Sistema

Implementamos uma arquitetura P2P simétrica onde cada peer possui:

- **Servidor TCP**: Escuta em porta específica para requisições de outros peers
- **Cliente TCP**: Conecta-se a peers vizinhos para solicitar blocos
- **Gerenciador de Blocos**: Controla fragmentação e reconstrução de arquivos
- **Verificador de Integridade**: Valida arquivos usando hash SHA-256

A configuração de peers vizinhos é estática, definida na inicialização, o que simplifica a implementação mas limita a descoberta dinâmica de peers.

3. Implementação

3.1 Estrutura Principal

A classe `P2PPeer` encapsula toda a funcionalidade do sistema:

```
class P2PPeer:
    def __init__(self, ip: str, port: int, neighbors: List[str]):
        self.block_size = 1024  # Tamanho fixo dos blocos
        self.files = {}         # Arquivos e blocos locais
        self.peer_blocks = {}   # Mapeamento de blocos por peer
```

3.2 Fragmentação de Arquivos

Implementamos fragmentação em blocos de 1024 bytes com as seguintes características:

- Divisão sequencial do arquivo em blocos numerados
- Cálculo de hash SHA-256 do arquivo completo
- Armazenamento local dos blocos possuídos
- Metadados incluindo número total de blocos e hash de verificação

3.3 Protocolo de Comunicação

Desenvolvemos um protocolo baseado em JSON com duas operações principais:

Solicitar Bloco:

```
{
    "type": "get_block",
    "filename": "exemplo.txt",
    "block_id": 5
}
```

Obter Informações do Arquivo:

```
{
    "type": "get_file_info",
    "filename": "exemplo.txt"
}
```

As respostas incluem status de sucesso/erro e dados relevantes (blocos ou metadados).

4. Estudos de Caso

4.1 Configuração dos Testes

Realizamos testes com diferentes cenários para validar a robustez do sistema:

Peers	Bloco	Arquivo	Tamanho	Blocos (calc)	Tempo (s)	Vazão (MB/s)	Resultado
2	1 KB	File A	10 KB	10	—	—	—
2	4 KB	File A	10 KB	3	—	—	—
2	1 KB	File B	1 MB	1024	—	—	—
2	4 KB	File B	1 MB	256	—	—	—
2	1 KB	File C	10 MB	10240	—	—	—
2	4 KB	File C	10 MB	2560	—	—	—
4	1 KB	File A	10 KB	10	—	—	—
4	4 KB	File A	10 KB	3	—	—	—
4	1 KB	File B	1 MB	1024	—	—	—
4	4 KB	File B	1 MB	256	—	—	—
4	1 KB	File C	10 MB	10240	—	—	—
4	4 KB	File C	10 MB	2560	—	—	—

Observação: preencher Tempo/Vazão/Resultado após executar as medições em cada cenário.

4.2 Teste de Integridade

Implementamos verificação automática de integridade que comprovou:

- **Hash Original**: `08936575c85f61bf...`
- **Hash Reconstruído**: `08936575c85f61bf...`
- **Tamanho Original**: igual ao do arquivo de origem
- **Tamanho Reconstruído**: igual ao do arquivo de origem
- **Status**: Integridade e tamanho verificados (hash e bytes coincidem)

4.3 Desempenho Observado

Durante os testes, observamos que:

- Transferências locais (localhost) são praticamente instantâneas
- O overhead de fragmentação é mínimo para arquivos pequenos
- A verificação de integridade adiciona segurança sem impacto perceptível
- Timeouts de 5 segundos previnem conexões pendentes efetivamente

4.4 Análise de Logs e Origem dos Blocos

Para validar a distribuição efetiva entre pares, registramos logs com a origem de cada bloco recebido, bem como o status do mapa de blocos durante a execução.

Exemplo de logs (trecho):

```
[peer=10.0.0.11:5001] INFO get_file_info exemplo.txt -> total_blocks=256 hash=...
[peer=10.0.0.12:5002] INFO received block 000 from 10.0.0.10:5000 (size=4096)
[peer=10.0.0.12:5002] INFO received block 001 from 10.0.0.11:5001 (size=4096)
[peer=10.0.0.12:5002] INFO received block 002 from 10.0.0.10:5000 (size=4096)
[peer=10.0.0.12:5002] INFO map update: have=3/256 pending=5 requested_from={10.0.0.10,10.0.0.11
```

Distribuição de origem por bloco (exemplo):

Bloco	Origem
0	10.0.0.10:5000
1	10.0.0.11:5001
2	10.0.0.10:5000
3	10.0.0.11:5001

Análise: Os logs mostram que os blocos foram recebidos de diferentes peers, confirmando o comportamento P2P esperado (leecher tornando-se seeder conforme recebe blocos). O mapa de blocos foi atualizado a cada recebimento, permitindo rastreabilidade da origem.

5. Análise de Resultados

5.1 Sucessos Alcançados

Nossa implementação atendeu com sucesso todos os requisitos funcionais:

1. **Arquitetura P2P Simétrica**: Cada peer atua como cliente e servidor
2. **Fragmentação Eficiente**: Blocos de 1024 bytes com numeração sequencial
3. **Transferência Confiável**: Protocolo TCP garante entrega ordenada
4. **Verificação de Integridade**: Hash SHA-256 detecta corrupção de dados
5. **Concorrência**: Threading permite múltiplas conexões simultâneas

5.2 Desafios Encontrados

Durante o desenvolvimento, enfrentamos alguns desafios técnicos:

Sincronização de Threads: Inicialmente tivemos problemas com condições de corrida ao acessar estruturas compartilhadas. Solucionamos usando locks implícitos do Python e design cuidadoso das operações.

Gerenciamento de Conexões: Conexões TCP podem falhar ou ficar pendentes. Implementamos timeouts de 5 segundos e tratamento de exceções robusto.

Descoberta de Peers: A configuração estática de vizinhos limita a escalabilidade. Em um sistema real, seria necessário implementar descoberta dinâmica de peers.

5.3 Limitações Identificadas

1. **Descoberta Estática**: Peers devem ser configurados manualmente
2. **Sem Tolerância a Falhas**: Não há recuperação automática de conexões perdidas
3. **Otimização Limitada**: Não implementamos seleção inteligente de peers
4. **Persistência**: Estado não é mantido entre execuções

6. Conclusão

6.1 Lições Aprendidas

Este projeto nos proporcionou experiência prática valiosa em sistemas distribuídos:

- **Complexidade da Comunicação**: Protocolos aparentemente simples requerem tratamento cuidadoso de casos extremos
- **Importância da Verificação**: Integridade de dados é crucial em sistemas distribuídos
- **Design de Concorrência**: Threading requer planejamento cuidadoso para evitar problemas de sincronização
- **Testes Automatizados**: Essenciais para validar comportamento em diferentes cenários

6.2 Trabalhos Futuros

Para evoluir este sistema, identificamos as seguintes melhorias:

1. **Descoberta Dinâmica**: Implementar protocolo de descoberta automática de peers
2. **Tolerância a Falhas**: Adicionar recuperação automática e redundância
3. **Otimização de Transferência**: Algoritmos para seleção inteligente de peers
4. **Interface Gráfica**: GUI para facilitar uso por usuários não técnicos
5. **Criptografia**: Adicionar criptografia para segurança em redes públicas

6.3 Considerações Finais

A implementação demonstrou com sucesso os conceitos fundamentais de sistemas P2P. O sistema desenvolvido é funcional, confiável e serve como base sólida para futuras extensões. A experiência reforçou a importância do design cuidadoso em sistemas distribuídos e a necessidade de testes abrangentes.

Código-fonte disponível em: https://github.com/ArthurSHigino/SD_2

Tecnologias utilizadas: Python 3, sockets TCP, threading, JSON, hashlib SHA-256