# **Conceitos Fundamentais de Sistemas Distribuídos - MiniBit**

## **1. 📊 DIVISÃO DE ARQUIVOS EM BLOCOS**

### **O que é?**

É o processo de quebrar um arquivo grande em pedaços menores, numerados sequencialmente, para facilitar o compartilhamento e download.

### **Por que fazer?**

* **Paralelismo**: Múltiplas pessoas podem baixar pedaços diferentes simultaneamente
* **Tolerância a falhas**: Se um pedaço corrompe, só precisa baixar aquele pedaço novamente
* **Eficiência**: Não precisa esperar o arquivo inteiro para começar a usar partes dele

### **Exemplo do Cotidiano: Pizza Delivery**

Imagine que você quer entregar uma pizza gigante para 10 pessoas:

* **Sem divisão**: Precisa carregar a pizza inteira de uma vez (pesado, difícil)
* **Com divisão**: Corta em 10 fatias e cada entregador leva algumas fatias (rápido, paralelo)

### **Técnicas de Implementação:**

# Exemplo simplificado

def dividir\_arquivo(arquivo, tamanho\_bloco=256\*1024): # 256KB por bloco

blocos = []

with open(arquivo, 'rb') as f:

contador = 0

while True:

pedaco = f.read(tamanho\_bloco)

if not pedaco:

break

blocos.append({

'id': contador,

'dados': pedaco,

'hash': calcular\_hash(pedaco) # Para verificar integridade

})

contador += 1

return blocos

### **Exemplos Comuns:**

* **YouTube**: Vídeos são divididos em segmentos de poucos segundos
* **WhatsApp**: Arquivos grandes são enviados em chunks
* **Netflix**: Filmes são divididos em segmentos para streaming adaptativo

## **2. 🤝 COMPARTILHAMENTO PEER-TO-PEER (P2P)**

### **O que é?**

Sistema onde cada participante (peer) atua simultaneamente como cliente (baixa dados) e servidor (fornece dados), eliminando a necessidade de um servidor central.

### **Como funciona?**

Cada peer mantém uma lista de outros peers e se conecta diretamente com eles para trocar dados.

### **Exemplo do Cotidiano: Biblioteca Comunitária**

Imagine um grupo de estudantes trocando livros:

* **Modelo tradicional**: Todos vão à biblioteca central (gargalo, ponto único de falha)
* **Modelo P2P**: Cada pessoa tem alguns livros em casa e empresta diretamente para os outros

### **Técnicas de Implementação:**

# Cada peer precisa ser servidor e cliente

class Peer:

def \_\_init\_\_(self, id, porta):

self.id = id

self.servidor = iniciar\_servidor(porta) # Serve arquivos

self.cliente = ClienteP2P() # Baixa arquivos

self.peers\_conhecidos = []

def baixar\_bloco(self, bloco\_id):

for peer in self.peers\_conhecidos:

if peer.tem\_bloco(bloco\_id):

return self.cliente.baixar\_de(peer, bloco\_id)

def servir\_bloco(self, bloco\_id):

if self.tenho\_bloco(bloco\_id):

return self.meus\_blocos[bloco\_id]

### **Vantagens:**

* **Escalabilidade**: Quanto mais peers, maior a capacidade total
* **Resistência a falhas**: Se alguns peers saem, outros continuam funcionando
* **Eficiência de custos**: Não precisa de servidores caros

### **Exemplos Comuns:**

* **BitTorrent**: Compartilhamento de arquivos
* **Skype** (versões antigas): Chamadas diretas entre usuários
* **Blockchain**: Cada nó mantém uma cópia do ledger

## **3. 🏢 TRACKER CENTRAL**

### **O que é?**

Um servidor centralizado que funciona como "catálogo telefônico" - não armazena os arquivos, mas sabe quem tem o quê.

### **Para que serve?**

* **Descoberta de peers**: "Quem tem o arquivo X?"
* **Bootstrap da rede**: Como novos peers encontram a rede existente
* **Coordenação básica**: Mantém lista atualizada de peers ativos

### **Exemplo do Cotidiano: Lista Telefônica**

* **Você quer**: Pizza
* **Lista telefônica**: "Pizzarias: João (123-456), Maria (789-012), Pedro (345-678)"
* **Você liga diretamente** para as pizzarias (não para a lista telefônica)

### **Técnicas de Implementação:**

class Tracker:

def \_\_init\_\_(self):

self.peers\_ativos = {} # {peer\_id: {ip, porta, blocos}}

def registrar\_peer(self, peer\_id, ip, porta):

self.peers\_ativos[peer\_id] = {

'ip': ip, 'porta': porta, 'ultimo\_visto': agora()

}

def obter\_peers\_para(self, peer\_solicitante):

# Retorna lista de outros peers (exceto o solicitante)

outros\_peers = [p for p in self.peers\_ativos

if p != peer\_solicitante]

return random.sample(outros\_peers, min(5, len(outros\_peers)))

### **Limitações:**

* **Ponto único de falha**: Se tracker cai, novos peers não conseguem entrar
* **Gargalo potencial**: Todos os peers consultam o mesmo servidor
* **Alvo de censura**: Governos podem bloquear o tracker

### **Exemplos Comuns:**

* **DNS**: Resolve nomes (google.com) para IPs (172.217.14.46)
* **WhatsApp Web**: QR code conecta celular com navegador
* **Uber**: App encontra motoristas próximos

## **4. 🔍 ALGORITMO RAREST FIRST**

### **O que é?**

Estratégia que prioriza o download dos blocos mais raros (que poucos peers possuem) primeiro.

### **Por que é importante?**

* **Evita extinção**: Garante que blocos raros não desapareçam da rede
* **Melhora disponibilidade**: Distribui blocos raros para mais peers
* **Otimiza a rede**: Mantém diversidade de conteúdo

### **Exemplo do Cotidiano: Coleção de Cartas**

Imagine que você coleciona cartas de futebol com seus amigos:

* **Carta comum**: Jogador X (todo mundo tem)
* **Carta rara**: Jogador Y (só 2 pessoas têm)
* **Estratégia inteligente**: Priorize conseguir a carta rara primeiro, porque se essas 2 pessoas pararem de colecionar, a carta pode "sumir" para sempre

### **Técnicas de Implementação:**

def calcular\_raridade(self, blocos\_disponiveis):

# Conta quantos peers têm cada bloco

contador\_blocos = {}

for peer in self.peers\_conhecidos:

for bloco in peer.blocos:

contador\_blocos[bloco] = contador\_blocos.get(bloco, 0) + 1

# Ordena por raridade (menos comum primeiro)

blocos\_por\_raridade = sorted(blocos\_disponiveis.keys(),

key=lambda b: contador\_blocos.get(b, 0))

return blocos\_por\_raridade

def proximo\_bloco\_para\_baixar(self):

blocos\_que\_preciso = self.blocos\_faltantes()

blocos\_disponiveis = self.blocos\_nos\_peers\_conhecidos()

# Intersecção: blocos que preciso E estão disponíveis

candidatos = set(blocos\_que\_preciso) & set(blocos\_disponiveis)

# Prioriza os mais raros

return self.calcular\_raridade(candidatos)[0]

### **Exemplo Prático:**

Suponha uma rede com 5 peers e 10 blocos:

* Bloco 0: 4 peers têm (comum)
* Bloco 5: 1 peer tem (MUITO RARO) ← **Prioridade máxima**
* Bloco 7: 2 peers têm (raro)
* Bloco 3: 3 peers têm (pouco comum)

### **Benefícios:**

* **Preservação**: Blocos raros se espalham antes de desaparecer
* **Eficiência da rede**: Todos os peers contribuem com diversidade
* **Robustez**: Sistema funciona mesmo com peers saindo

## **5. ⚖️ TIT-FOR-TAT (OLHO POR OLHO)**

### **O que é?**

Estratégia de reciprocidade onde você prioriza ajudar quem te ajuda, incentivando cooperação e desencorajando "parasitas" (free-riders).

### **Como funciona no BitTorrent?**

* **4 slots "unchoked"**: Você envia dados para os 4 peers que mais te enviam dados
* **1 slot "optimistic"**: A cada 10s, testa um peer aleatório (pode ser que ele tenha algo raro)
* **Peers "choked"**: Bloqueados para download (podem ver sua lista de blocos, mas não baixar)

### **Exemplo do Cotidiano: Cooperativa de Carona**

Imagine um grupo de colegas que se revezam para dar carona:

* **João sempre dá carona** para Maria, Pedro, Ana
* **Carlos nunca dá carona** para ninguém
* **Resultado**: João prioriza dar carona para quem também dá carona (Maria, Pedro, Ana)
* **Carlos fica sem carona** (tit-for-tat em ação)
* **Excepção**: Às vezes João dá carona para alguém novo (optimistic unchoke) para ver se essa pessoa vai cooperar

### **Implementação Simplificada no MiniBit:**

class TitForTat:

def \_\_init\_\_(self):

self.peers\_unchoked = [] # Máximo 4 peers priorizados

self.optimistic\_peer = None # 1 peer aleatório

self.historico\_trocas = {} # Quantos blocos cada peer me deu

def atualizar\_slots(self):

# A cada 10 segundos, recalcula quem desbloquear

# 1. Ordena peers por quantos blocos me deram

peers\_ordenados = sorted(self.peers\_conhecidos,

key=lambda p: self.historico\_trocas.get(p.id, 0),

reverse=True)

# 2. Top 4 vão para slots fixos

self.peers\_unchoked = peers\_ordenados[:4]

# 3. Escolhe 1 peer aleatório para slot otimista

outros\_peers = peers\_ordenados[4:]

if outros\_peers:

self.optimistic\_peer = random.choice(outros\_peers)

def posso\_baixar\_de(self, peer):

return (peer in self.peers\_unchoked or

peer == self.optimistic\_peer)

### **Adaptação no MiniBit:**

Como estamos simulando em uma máquina (não podemos medir velocidade real), usamos "raridade dos blocos" em vez de "velocidade de upload":

* **Peers com blocos raros** são mais valiosos
* **Priorizamos peers** que têm blocos que poucos outros têm

### **Por que funciona?**

* **Incentiva cooperação**: Quem compartilha mais, recebe mais
* **Pune free-riders**: Quem só baixa sem compartilhar fica limitado
* **Auto-regulação**: Sistema se equilibra naturalmente
* **Explora novos peers**: Optimistic unchoke descobre novos cooperadores

## **6. 🔄 SINCRONIZAÇÃO E COORDENAÇÃO**

### **O que é?**

Mecanismos para garantir que múltiplos peers funcionem de forma coordenada, mesmo executando de forma independente e assíncrona.

### **Desafios:**

* **Concorrência**: Múltiplos peers fazendo coisas ao mesmo tempo
* **Latência de rede**: Mensagens demoram para chegar
* **Falhas**: Peers podem desligar a qualquer momento
* **Estados inconsistentes**: Cada peer pode ter uma "visão" diferente da rede

### **Exemplo do Cotidiano: Organizar um Churrasco**

Imagine 10 amigos organizando um churrasco via WhatsApp:

* **Problema**: Todos falam ao mesmo tempo, mensagens chegam fora de ordem
* **Coordenação necessária**: Quem traz o quê? Que horas? Onde?
* **Solução**: Uma pessoa (coordenador) coleta as informações e redistribui

### **Técnicas Usadas no MiniBit:**

#### **Heartbeat (Sinal de Vida)**

def enviar\_heartbeat(self):

while self.ativo:

try:

requests.post(f"{TRACKER\_URL}/heartbeat",

json={"peer\_id": self.id})

time.sleep(30) # A cada 30 segundos

except:

print("Tracker indisponível")

#### **Locks para Thread Safety**

class BlockManager:

def \_\_init\_\_(self):

self.lock = threading.Lock() # Garante acesso exclusivo

self.blocks = {}

def add\_block(self, block\_id, data):

with self.lock: # Só um thread por vez pode modificar

self.blocks[block\_id] = data

#### **Timeouts para Operações de Rede**

def baixar\_bloco(self, peer, block\_id):

try:

socket.settimeout(10) # Máximo 10 segundos

response = socket.recv(data)

return parse\_response(response)

except socket.timeout:

print(f"Peer {peer} não respondeu a tempo")

return None

### **Padrões de Coordenação:**

#### **1. Producer-Consumer (Produtor-Consumidor)**

* **Tracker produz** lista de peers
* **Peers consomem** a lista para descobrir outros peers

#### **2. Request-Response (Solicitação-Resposta)**

* Peer A solicita bloco X do Peer B
* Peer B responde com dados do bloco ou erro

#### **3. Publish-Subscribe (Publicar-Assinar)**

* Peers "publicam" quando recebem novos blocos
* Outros peers "assinam" essas notificações

## **7. 🛡️ TOLERÂNCIA A FALHAS**

### **O que é?**

Capacidade do sistema continuar funcionando mesmo quando alguns componentes falham.

### **Tipos de Falhas:**

1. **Crash de peer**: Peer desliga inesperadamente
2. **Partição de rede**: Peers ficam isolados em grupos
3. **Corrupção de dados**: Blocos corrompidos durante transmissão
4. **Sobrecarga**: Peer muito lento ou sobrecarregado

### **Exemplo do Cotidiano: Sistema de Transporte Urbano**

* **Sem tolerância**: Se um ônibus quebra, toda a linha para
* **Com tolerância**: Vários ônibus na mesma linha, se um quebra, outros continuam atendendo

### **Técnicas no MiniBit:**

#### **Replicação de Dados**

def baixar\_bloco\_com\_backup(self, block\_id):

peers\_com\_bloco = self.encontrar\_peers\_com\_bloco(block\_id)

for peer in peers\_com\_bloco:

try:

bloco = self.baixar\_de(peer, block\_id)

if self.verificar\_integridade(bloco):

return bloco

except:

continue # Tenta próximo peer

raise Exception("Nenhum peer conseguiu fornecer o bloco")

#### **Verificação de Integridade**

def verificar\_bloco(self, dados, hash\_esperado):

hash\_calculado = hashlib.sha256(dados).hexdigest()

return hash\_calculado == hash\_esperado

#### **Cleanup de Peers Inativos**

def limpar\_peers\_mortos(self):

agora = time.time()

peers\_ativos = []

for peer in self.peers\_conhecidos:

if agora - peer.ultimo\_contato < 300: # 5 minutos

peers\_ativos.append(peer)

self.peers\_conhecidos = peers\_ativos

#### **Retry com Backoff Exponencial**

def baixar\_com\_retry(self, peer, block\_id, max\_tentativas=3):

for tentativa in range(max\_tentativas):

try:

return self.baixar\_de(peer, block\_id)

except:

if tentativa < max\_tentativas - 1:

time.sleep(2 \*\* tentativa) # 1s, 2s, 4s...

else:

raise

## **8. 📈 MÉTRICAS E MONITORAMENTO**

### **O que medir?**

* **Progresso**: Quantos blocos cada peer tem
* **Performance**: Velocidade de download/upload
* **Saúde da rede**: Quantos peers ativos, distribuição de blocos
* **Eficiência**: Tempo para completar download

### **Exemplo do Cotidiano: Dashboard de Delivery**

Como apps de delivery mostram:

* Onde está seu pedido
* Tempo estimado
* Quantos entregadores ativos
* Taxa de sucesso das entregas

### **Implementação:**

class Metrics:

def \_\_init\_\_(self):

self.start\_time = time.time()

self.blocks\_downloaded = 0

self.blocks\_uploaded = 0

self.bytes\_transferred = 0

def get\_stats(self):

runtime = time.time() - self.start\_time

return {

'runtime\_seconds': runtime,

'download\_speed': self.blocks\_downloaded / runtime,

'upload\_speed': self.blocks\_uploaded / runtime,

'progress': len(self.owned\_blocks) / self.total\_blocks

}

## **🎯 RESUMO: Como Tudo Se Conecta**

1. **Arquivo é dividido em blocos** → Facilita compartilhamento paralelo
2. **Tracker coordena descoberta** → Peers encontram uns aos outros
3. **P2P elimina gargalos** → Download direto entre peers
4. **Rarest First** → Garante diversidade e preservação de conteúdo
5. **Tit-for-Tat** → Incentiva cooperação e pune free-riders
6. **Tolerância a falhas** → Sistema continua funcionando com falhas
7. **Coordenação** → Peers trabalham juntos de forma ordenada

### **Analogia Final: Construção Colaborativa**

Imagine que 100 pessoas querem construir uma casa juntas:

* **Divisão em blocos**: Cada pessoa constrói uma parte (fundação, paredes, teto)
* **Tracker**: Quadro de avisos que diz quem está fazendo o quê
* **P2P**: Pessoas se ajudam diretamente, sem coordenador central
* **Rarest First**: Priorizam partes que poucos sabem fazer
* **Tit-for-Tat**: Quem mais ajuda, mais recebe ajuda
* **Tolerância a falhas**: Se alguém desiste, outros assumem a parte
* **Resultado**: Casa construída mais rápido e de forma resiliente

Este é o poder dos sistemas distribuídos: **coordenação eficiente de recursos distribuídos para atingir um objetivo comum!**