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!**