# Explicação Detalhada do Projeto e Código: Simulador P2P

## Introdução e Objetivo

Este projeto consiste na implementação de um sistema de compartilhamento de arquivos Peer-to-Peer (P2P) em Python, concebido para simular aspectos fundamentais do protocolo BitTorrent. O objetivo principal é demonstrar o uso de sockets para comunicação em rede, especificamente UDP e TCP, na construção de uma aplicação distribuída. A aplicação permite que múltiplos clientes (peers) colaborem para distribuir um arquivo grande, coordenados por um servidor centralizado (tracker), sem que a fonte original precise enviar o arquivo completo para cada destino individualmente.

## Arquitetura Geral

A arquitetura do sistema é composta por dois tipos principais de componentes:

1. **Tracker (`tracker.py`):** Um servidor centralizado que atua como um ponto de coordenação. Ele não armazena o arquivo em si, mas mantém o registro de quais peers estão participando da rede (swarm) e quais pedaços (pieces) do arquivo cada peer possui. Sua função é facilitar a descoberta de peers.
2. **Peer (`peer.py`):** Um cliente que representa um nó na rede P2P. Cada peer pode atuar tanto como cliente (baixando pedaços de outros peers) quanto como servidor (enviando pedaços para outros peers). Peers se comunicam com o Tracker para entrar na rede e anunciar os pedaços que possuem, e se comunicam diretamente entre si para trocar os dados do arquivo.

## O Tracker (`tracker.py`)

O script `tracker.py` implementa o servidor Tracker. Sua principal responsabilidade é gerenciar o estado da rede P2P. Ele escuta por mensagens UDP em uma porta específica (configurada como 10000 por padrão). A escolha do UDP para a comunicação com o Tracker visa a leveza e a escalabilidade, já que as interações são geralmente curtas e não exigem a garantia de entrega ou a sobrecarga de uma conexão TCP.

Internamente, o Tracker mantém uma estrutura de dados central, `peers_db`, que é um dicionário Python. As chaves deste dicionário são identificadores únicos de peers

(formados por `ip:porta_tcp`), e os valores são outros dicionários contendo informações sobre cada peer: seu endereço IP, a porta TCP onde ele escuta por conexões P2P, um conjunto (`set`) com os índices dos pedaços do arquivo que ele declarou possuir, e um timestamp da última vez que enviou uma atualização. Para garantir que o acesso a `peers_db` seja seguro em um ambiente concorrente (embora o modelo atual processe UDP sequencialmente, uma trava foi incluída por boas práticas), utiliza-se um `threading.Lock` chamado `db_lock`.

O fluxo principal do Tracker ocorre na função `start_tracker`. Ela cria um socket UDP, o vincula ao endereço e porta configurados, e entra em um loop infinito aguardando mensagens. Cada mensagem recebida é processada pela função `handle_udp_message`. Esta função analisa o comando da mensagem:

- **JOIN:** Quando um novo peer entra na rede, ele envia uma mensagem `JOIN <seu_ip> <sua_porta_tcp>`. O Tracker registra o novo peer em `peers_db` (inicialmente sem nenhum pedaço) e responde ao peer com uma mensagem `PEERLIST`, contendo a lista formatada de todos os outros peers atualmente conhecidos e seus respectivos pedaços. A formatação da lista é feita pela função `format_peer_list`.
- **UPDATE:** Periodicamente (a cada 30 segundos, conforme definido no `peer.py`), cada peer envia uma mensagem `UPDATE <seu_ip> <sua_porta_tcp> [<lista_de_indices_de_pedacos>]`. O Tracker utiliza essa mensagem para atualizar o conjunto de pedaços do peer correspondente em `peers_db` e também atualiza o timestamp `last_update`.

Além disso, o Tracker possui uma thread secundária, iniciada em `start_tracker` e executando a função `cleanup_inactive_peers`. Esta thread periodicamente verifica `peers_db` e remove os peers cujo `last_update` é mais antigo que um tempo limite definido (`PEER_TIMEOUT`), garantindo que a lista de peers permaneça relativamente atualizada e livre de nós que saíram da rede sem avisar.

## O Peer (`peer.py`)

O script `peer.py` implementa o cliente Peer, que é a parte mais complexa do sistema, pois lida com múltiplas responsabilidades e protocolos de comunicação.

**Inicialização e Configuração:** Ao ser executado, o `peer.py` utiliza `argparse` para receber argumentos da linha de comando, como o caminho para o arquivo alvo, o endereço do Tracker, a porta TCP em que o próprio peer escutará, e o caminho para o arquivo de saída onde o download será salvo. Ele determina seu próprio IP (usando uma técnica de conexão UDP a um servidor externo como 8.8.8.8) e calcula o número total de

pedaços ( total_pieces ) com base no tamanho do arquivo alvo e no tamanho definido para cada pedaço ( PIECE_SIZE ). Crucialmente, ele inicializa o arquivo de saída ( initialize_output_file ), pré-alocando o espaço necessário em disco (preenchendo com bytes nulos) para que os pedaços possam ser escritos diretamente em suas posições corretas posteriormente.

**Comunicação com o Tracker:** O peer cria um socket UDP para se comunicar com o Tracker. A primeira ação é enviar uma mensagem JOIN . Após o envio, ele aguarda a resposta PEERLIST do Tracker. A string recebida é processada pela função parse_peer_list para popular a estrutura de dados local known_peers , que espelha parcialmente a peers_db do Tracker, mas focada nos outros peers. A função update_known_peers gerencia a atualização segura dessa estrutura local usando um threading.Lock ( state_lock ).

**Comunicação Peer-to-Peer (TCP):** Para a troca efetiva dos pedaços do arquivo, os peers utilizam o protocolo TCP, que garante a entrega ordenada e confiável dos dados. Cada peer precisa ser capaz de:

1. **Atuar como Servidor:** Uma thread dedicada ( tcp_server_thread ) é iniciada para escutar na porta TCP especificada ( --listen-port ). Quando outro peer se conecta, a função handle_peer_connection (executada em uma nova thread para cada conexão) recebe a solicitação. Se for uma mensagem GET <indice_pedaco> , o peer verifica se possui o pedaço solicitado (consultando o conjunto owned_pieces ). Se possuir, ele lê os dados do pedaço do arquivo local usando read_piece e os envia pela conexão TCP. A função read_piece calcula o offset correto no arquivo e lê o número apropriado de bytes ( PIECE_SIZE ou menos para o último pedaço).
2. **Atuar como Cliente:** A lógica principal de download, gerenciada pela função download_manager , é responsável por iniciar conexões TCP com outros peers para baixar os pedaços necessários. O processo de escolha é crucial e segue a estratégia "rarest-first".

**Estratégia Rarest-First e Download:** A função calculate_rarity analisa a lista known_peers para contar quantos peers possuem cada pedaço do arquivo. A função choose_piece_to_download então identifica quais pedaços o peer atual ainda não possui ( needed_pieces ), ordena esses pedaços pelo número de peers que os têm (do menor para o maior, ou seja, do mais raro para o mais comum), e seleciona o mais raro. Em seguida, identifica quais peers na lista known_peers possuem esse pedaço raro e escolhe aleatoriamente um deles para tentar o download. A função download_piece estabelece a conexão TCP com o peer escolhido, envia a mensagem GET <indice_pedaco> , recebe os dados do pedaço em um loop (para garantir que todo o pedaço seja recebido, mesmo que chegue em múltiplos pacotes TCP), verifica se o

tamanho está correto, e então utiliza a função `write_piece` para salvar os dados na posição correta do arquivo de saída pré-alocado. Após baixar um pedaço com sucesso, o índice é adicionado ao conjunto `owned_pieces`, e uma mensagem `UPDATE` é imediatamente enviada ao Tracker para anunciar a posse do novo pedaço.

**Concorrência e Threads:** O `peer.py` utiliza `threading` extensivamente para lidar com tarefas simultâneas:

- A thread `tcp_server_thread` escuta por conexões TCP de entrada.
- A thread `tracker_update_thread` envia atualizações UDP para o Tracker a cada `UPDATE_INTERVAL` segundos.
- A lógica de download (`download_manager`) roda (nesta implementação) na thread principal, mas poderia ser movida para sua própria thread. Ela coordena a escolha e o download de peças.
- Cada conexão TCP de entrada para upload é tratada em sua própria thread por `handle_peer_connection`.

O uso de `state_lock` é essencial para proteger o acesso concorrente às estruturas de dados compartilhadas (`known_peers` e `owned_pieces`) por essas diferentes threads, evitando condições de corrida.

**Gerenciamento de Arquivo:** O arquivo é tratado como uma sequência de pedaços de tamanho fixo (`PIECE_SIZE`). Funções como `get_piece_size`, `read_piece`, e `write_piece` abstraem o acesso ao arquivo com base no índice do pedaço, calculando os offsets e tamanhos corretos, incluindo o tratamento especial para o último pedaço, que pode ser menor.

## Conclusão

O projeto implementa com sucesso uma simulação funcional de um sistema P2P básico usando sockets Python. Ele demonstra a interação entre um Tracker centralizado (UDP) e Peers distribuídos que colaboram para compartilhar um arquivo usando comunicação direta (TCP). A implementação aborda conceitos importantes como comunicação UDP e TCP, threading para concorrência, gerenciamento de estado distribuído (listas de peers e pedaços), estratégias de seleção de pedaços (rarest-first) e manipulação de arquivos binários em pedaços. O código está estruturado em dois scripts principais, com funções bem definidas para cada responsabilidade, e utiliza logging para facilitar o acompanhamento e a depuração.