Documentação Completa do Projeto MiniBit

Arquivo: `main.py`

Este é o ponto de entrada do programa. Ele é responsável por interpretar os argumentos da linha de comando e iniciar o Tracker ou um Peer.

- **`main()`**
  - **Função**: Orquestra a execução do programa.
  - **Detalhes**: Utiliza a biblioteca `argparse` para criar dois subcomandos: `tracker` e `peer`. Com base no comando fornecido pelo usuário, ele instancia e inicia o objeto correspondente (`Tracker` ou `Peer`). Para os peers, ele também lida com a configuração inicial, seja como "seeder" (usando `--file-path`) ou como "leecher" (usando `--file-name`). Ele também gerencia o ciclo de vida da aplicação, mantendo-a ativa e lidando com o encerramento gracioso via `Ctrl+C`.

Arquivo: `minibit/tracker.py`

Contém a lógica do servidor central que coordena a descoberta de peers.

Classe: `Tracker`

- **`__init__(self, host, port)`**
  - [cite_start]**Função**: Construtor da classe `Tracker`.
  - [cite_start]**Detalhes**: Inicializa as variáveis essenciais, como o endereço do host e a porta, a estrutura de dados `self.peers` que armazenará as informações dos peers conectados para cada arquivo, e um `threading.Lock` para garantir que o acesso a essa estrutura seja seguro em um ambiente com múltiplas threads.

- **`start(self)`**
  - **Função**: Inicia o servidor do tracker.
  - [cite_start]**Detalhes**: Cria e configura o socket do servidor, o coloca em modo de escuta e dispara uma nova thread para o método `_accept_connections`. Isso evita que o processo principal seja bloqueado.

- **`stop(self)`**
  - **Função**: Para o servidor do tracker de forma graciosa.
  - **Detalhes**: Altera a flag `self.running` para `False` e fecha o socket do servidor. Inclui um truque de se conectar ao próprio socket para desbloquear o `accept()` que estaria em espera, permitindo que a thread do servidor termine corretamente.

- **`_accept_connections(self)`**
  - **Função**: Loop principal para aceitar novas conexões de peers.
  - [cite_start]**Detalhes**: Fica em um loop infinito (enquanto `self.running` for `True`), esperando por novas conexões. [cite_start]Para cada conexão aceita, ele cria e inicia uma nova thread para o método `_handle_client` para processar as requisições daquele peer.

- **`_handle_client(self, conn, addr)`**
  - [cite_start]**Função**: Lida com toda a comunicação de um único peer conectado.
  - **Detalhes**: Recebe e processa mensagens em um loop. Lê o tamanho da mensagem, depois a mensagem em si (que é um JSON), e a envia para `_process_command`. Envia a resposta de volta para o peer. [cite_start]Contém a lógica de tratamento de erros para desconexões e remove o peer da lista caso a conexão seja perdida de forma inesperada.

- **`_process_command(self, message)`**
  - **Função**: "Cérebro" do tracker; processa os comandos recebidos.
  - [cite_start]**Detalhes**: Recebe a mensagem desserializada e, com base no valor da chave `"command"`, executa a ação apropriada (`REGISTER`, `GET_PEERS`, `UPDATE_BLOCKS`). Todas as modificações na estrutura de dados `self.peers` são protegidas por um lock.
    - [cite_start] **`REGISTER`**: Adiciona um peer e os blocos que ele possui à lista de um determinado arquivo.
    - [cite_start] **`GET_PEERS`**: Retorna uma lista de outros peers que possuem o arquivo solicitado.
    - [cite_start] **`UPDATE_BLOCKS`**: Atualiza a lista de blocos de um peer já registrado.

- **`_remove_peer(self, peer_id_to_remove)`**
  - **Função**: Remove um peer de todos os registros.
  - [cite_start]**Detalhes**: É chamado quando uma conexão com um peer falha, garantindo que peers "mortos" não sejam listados para outros. [cite_start]Ele varre todos os arquivos e remove o ID do peer de onde quer que ele apareça.

Arquivo: `minibit/peer.py`

Contém a implementação principal do nó P2P, que pode atuar tanto como seeder quanto como leecher.

Classe: `Peer`

- `__init__(self, tracker_host, tracker_port, listen_port)`
  - [cite_start]**Função**: Construtor da classe `Peer`.
  - [cite_start]**Detalhes**: Inicializa o ID único do peer, o endereço do tracker, as estruturas de dados para gerenciar conexões (`self.connections`) e informações de peers conhecidos (`self.known_peers_info`), e instancia os gerenciadores `BlockManager` e `UnchokeManager`.

- `start(self)`
  - [cite_start]**Função**: Inicia o peer e suas threads de background.
  - **Detalhes**: Cria o socket do servidor do peer para escutar conexões de outros peers e inicia as threads principais:
    1. `_accept_connections`: Para aceitar conexões de entrada.
    2. `_manage_connections_and_requests`: Para se conectar a outros e pedir blocos.
    3. [cite_start] `_run_unchoke_logic`: Para executar a estratégia "olho por olho".

- `stop(self)`
  - [cite_start]**Função**: Para todas as operações do peer.
  - [cite_start]**Detalhes**: Altera a flag `self.running` e fecha todas as conexões ativas e o socket do servidor.

- `share_file(self, file_path, block_size)`
  - **Função**: Configura o peer para atuar como seeder.
  - **Detalhes**: Instancia o `BlockManager`, carrega o arquivo do disco usando `load_from_file`, e se registra no tracker como possuidor de 100% dos blocos.

- `download_file(self, file_name, block_size)`
  - **Função**: Configura o peer para atuar como leecher.
  - [cite_start]**Detalhes**: Instancia o `BlockManager` (sem blocos), se registra no tracker (com 0 blocos) e imediatamente pede uma lista de peers para começar o download.

- `is_download_complete(self)`
  - **Função**: Verifica se o peer já baixou todos os blocos.
  - **Detalhes**: Simplesmente delega a chamada para o método `is_complete()` do `BlockManager`.

- `_accept_connections(self)` e `_handle_incoming_connection(self)`
  - [cite_start]**Função**: Lidam com conexões de entrada de outros peers.
  - **Detalhes**: `_accept_connections` espera por novas conexões. [cite_start] `_handle_incoming_connection` realiza o "handshake" (troca inicial de IDs), adiciona a nova conexão à lista e inicia o loop de mensagens (`_message_loop`) para aquele peer.

- `_connect_to_peer(self, peer_id, address)`
  - [cite_start]**Função**: Estabelece uma conexão de saída para outro peer.
  - [cite_start]**Detalhes**: Cria um objeto `PeerConnection`, conecta-se ao endereço fornecido, realiza o handshake e, se bem-sucedido, inicia o `_message_loop`.

- `_message_loop(self, peer_conn, peer_id)`
  - **Função**: Loop de processamento de mensagens para uma única conexão P2P.
  - **Detalhes**: É o coração da comunicação P2P. Fica em um loop lendo mensagens e agindo de acordo com o tipo (`have`, `request_block`, `block_data`, `choke`, `unchoke`). [cite_start]Lida com a recepção de blocos, o envio de blocos requisitados e as atualizações de estado de choke.

- `_manage_connections_and_requests(self)`
  - **Função**: Thread periódica que gerencia o download.
  - **Detalhes**: A cada 5 segundos, ela tenta se conectar a novos peers obtidos do tracker e chama `_request_blocks` para tentar baixar novas peças do arquivo. [cite_start]Também é responsável por logar o status do progresso.

- `_request_blocks(self)`
  - [cite_start]**Função**: Implementa a lógica "Rarest First".
  - [cite_start]**Detalhes**: Pede ao `BlockManager` a lista dos blocos faltantes mais raros. [cite_start]Então, para o bloco mais raro, encontra um peer que o possua e que não esteja nos "choking" (bloqueando) e envia uma mensagem `request_block`.

- `_run_unchoke_logic(self)`
  - **Função**: Thread periódica que implementa a lógica "Olho por Olho".
  - [cite_start]**Detalhes**: A cada 10 segundos, identifica quais peers conectados estão interessados em seus blocos. [cite_start]Passa essa lista para o

`UnchokeManager` para obter a decisão de quem deve receber `choke` e `unchoke`, e então envia as mensagens apropriadas.

- `_send_to_tracker(self, message)` e funções relacionadas
  - **Função**: Métodos que encapsulam a comunicação com o tracker.
  - **Detalhes**: `_send_to_tracker` abre uma conexão temporária com o tracker, envia uma mensagem e espera pela resposta. `_register_with_tracker` e `_update_peers_from_tracker` usam esta função para executar suas respectivas lógicas.

Arquivo: `minibit/block_manager.py`

[cite_start]Responsável por toda a lógica relacionada aos blocos de um arquivo.

Classe: `BlockManager`

- `__init__(self, file_name, block_size, logger)`
  - **Função**: Construtor da classe.
  - [cite_start]**Detalhes**: Inicializa as estruturas de dados para guardar os blocos que o peer possui (`self.my_blocks`) e para rastrear quais blocos outros peers têm (`self.peer_block_map`), o que é essencial para o "rarest first".

- `load_from_file(self, file_path)`
  - [cite_start]**Função**: Lê um arquivo do disco e o divide em blocos.
  - **Detalhes**: Usado pelo seeder. [cite_start]Abre o arquivo, lê em pedaços (`block_size`) e armazena cada pedaço em memória no dicionário `self.my_blocks` .

- `reconstruct_file(self)`
  - [cite_start]**Função**: Remonta o arquivo a partir dos blocos baixados.
  - **Detalhes**: Usado pelo leecher ao final do download. [cite_start]Ordena os blocos pelo seu índice e os escreve em sequência em um novo arquivo no diretório `downloads/` .

- `add_block(self, block_id, data)`
  - [cite_start]**Função**: Armazena um novo bloco recebido de outro peer.

- `get_rarest_missing_blocks(self)`
  - [cite_start]**Função**: Retorna os blocos faltantes, ordenados do mais raro ao mais comum.
  - **Detalhes**: O "cérebro" da estratégia "rarest first". [cite_start]Ele usa o `self.peer_block_map` para calcular a contagem de cada bloco na rede e ordena a lista de blocos que faltam com base nessa contagem.

- `update_peer_blocks(self, peer_id, their_blocks)`
  - **Função**: Atualiza o mapa de raridade.
  - [cite_start]**Detalhes**: Chamado sempre que um peer informa quais blocos possui (via mensagem `have`). [cite_start]Ele atualiza a estrutura `self.peer_block_map` para refletir o estado atual da rede.

- `get_peer_blocks(self, peer_id)`
  - **Função**: Retorna o conjunto de blocos que um peer específico possui.
  - **Detalhes**: Método auxiliar usado pela lógica de "unchoke" para determinar se um peer está interessado.

Arquivo: `minibit/unchoke_manager.py`

[cite_start]Implementa a estratégia "Olho por Olho" (tit-for-tat) simplificada.

Classe: `UnchokeManager`

- `__init__(self, my_peer_id, logger)`
  - [cite_start]**Função**: Construtor da classe.
  - [cite_start]**Detalhes**: Inicializa os conjuntos que guardarão os IDs dos peers que estão com "unchoke" (os fixos e o otimista).

- `evaluate_peers(self, interested_peers, block_rarity)`
  - [cite_start]**Função**: Decide quem deve receber `choke` e `unchoke` .
  - **Detalhes**: Esta é a função principal. Recebe la lista de peers interessados, embaralha-a aleatoriamente e seleciona os primeiros 4 como "fixos" e o

próximo como "otimista". Compara essa nova lista com a do ciclo anterior para determinar quem deve ser alterado de `choke` para `unchoke` e vice-versa. [cite_start]Retorna duas listas: `to_choke` e `to_unchoke`.

- `is_unchoked(self, peer_id)`
  - [cite_start]**Função**: Verifica se um peer específico pode receber upload.
  - **Detalhes**: Usado pelo `Peer` para decidir se deve ou não responder a um `request_block`.

Arquivo: `minibit/peer_connection.py`

[cite_start]Abstrai a comunicação de baixo nível com um único peer.

Classe: `PeerConnection`

- `__init__(self, address, logger, sock)`
  - [cite_start]**Função**: Construtor da classe.
  - [cite_start]**Detalhes**: Armazena o endereço do peer e pode opcionalmente receber um socket já conectado (para conexões de entrada).

- `connect(self)`
  - [cite_start]**Função**: Conecta-se a um peer (para conexões de saída).

- `send_message(self, message)` e `read_message(self)`
  - [cite_start]**Função**: Lidam com o envio e recebimento de dados.
  - **Detalhes**: Implementam o protocolo de tamanho+payload. `send_message` serializa o dicionário para JSON, calcula o tamanho, envia o tamanho como 4 bytes e depois envia o JSON. [cite_start] `read_message` faz o processo inverso.

- `close(self)`
  - [cite_start]**Função**: Fecha o socket da conexão.

- `set_choked_by_peer(self, status)` e `is_choked_by_peer(self)`
  - **Função**: Gerenciam o estado de "choke" imposto pelo outro peer.
  - **Detalhes**: Um peer leecher usa `is_choked_by_peer` para saber se pode ou não requisitar um bloco de um seeder. [cite_start]O estado é alterado ao receber as mensagens `choke` e `unchoke`.