UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ARTES, CIÊNCIAS E HUMANIDADES

Nome: Felipe Gabriel Gomes de Carvalho

N° USP: 13719180

Relatório de Implementação: Sistema Peer-to-Peer (EP1)

Disciplina: Desenvolvimento de Sistemas Distribuídos

Docente: Prof. Dr. Renan Cerqueira Afonso Alves

São Paulo
2025

1. Introdução

Este relatório documenta a implementação de um sistema peer-to-peer (P2P) básico desenvolvido em Python como parte do primeiro exercício programa (EP1). O sistema implementa os fundamentos de uma rede P2P onde cada nó (peer) pode se comunicar com outros nós para compartilhamento de informações e descoberta de pares na rede.

1.1 Contexto

A arquitetura peer-to-peer é um modelo de comunicação distribuída onde cada nó na rede atua tanto como cliente quanto como servidor, diferentemente do modelo cliente-servidor tradicional. Neste trabalho, implementamos um sistema onde:

- Cada peer é identificado por um endereço IP e porta
- Os peers mantêm uma lista de outros peers conhecidos
- A comunicação é realizada via sockets TCP
- Um protocolo de mensagens simples é utilizado para interação entre os nós
- Um relógio lógico (Lamport clock) é implementado para ordenação parcial de eventos

1.2 Objetivos

O código implementado atende aos seguintes requisitos funcionais:

1. Inicialização do peer com parâmetros de configuração (endereço, arquivo de vizinhos, diretório compartilhado)
2. Comunicação básica entre peers usando mensagens HELLO, GET_PEERS e BYE
3. Manutenção de uma lista de peers conhecidos com seus status (ONLINE/OFFLINE)
4. Implementação de um relógio lógico para ordenação de eventos
5. Interface de usuário com menu interativo para execução de comandos
6. Compartilhamento básico de arquivos (listagem local)

1.3 Tecnologias Utilizadas

- **Linguagem Python**: Escolhida por sua simplicidade e bibliotecas robustas para programação de redes
- **Sockets TCP**: Para comunicação entre os peers
- **Threading**: Para permitir operação concorrente do servidor e interface de usuário
- **Mecanismos de sincronização**: Locks para acesso thread-safe a estruturas de dados compartilhadas

1.4 Organização do Relatório

Este relatório está organizado da seguinte forma:

1. **Introdução**: Contextualização do trabalho e objetivos
2. **Arquitetura do Sistema**: Visão geral da estrutura do código e componentes principais
3. **Implementação**: Detalhes das principais funcionalidades implementadas
4. **Análise e Discussão**: Avaliação das decisões de implementação e possíveis melhorias
5. **Testes:** Execução de testes de comandos implementados.
6. **Conclusão**: Considerações finais sobre o trabalho desenvolvido

O código completo foi desenvolvido em um único arquivo Python (`eachare.py`) contendo todas as funcionalidades básicas exigidas na especificação do EP1, com extensibilidade para as próximas fases do projeto.

2. Arquitetura do Sistema

A arquitetura do sistema peer-to-peer desenvolvido é baseada em um modelo descentralizado, onde cada nó (**peer**) atua tanto como **cliente** quanto como **servidor**, permitindo comunicação direta entre os participantes da rede. A seguir, detalhamos os principais componentes e a estrutura do sistema.

2.1 Componentes Principais

2.1.1 Peer (Nó Individual)

Cada peer é identificado por um endereço único no formato `<IP>:<porta>` e possui as seguintes responsabilidades:

- **Manter uma lista de peers conhecidos** (com status ONLINE/OFFLINE).
- **Escutar conexões TCP** em sua porta designada para receber mensagens de outros peers.
- **Enviar mensagens** para outros peers conforme comandos do usuário.
- **Gerenciar um relógio lógico** (Lamport clock) para ordenação parcial de eventos.
- **Armazenar arquivos compartilhados** em um diretório local.

2.1.2 Threads Concorrentes

Para permitir operação simultânea do servidor e da interface do usuário, o sistema utiliza **threads**:

1. **Thread do Servidor (`server_thread`)**
    - Fica em execução contínua, aguardando conexões de outros peers.
    - Para cada conexão recebida, inicia uma **thread de tratamento (`handle_client`)** para processar a mensagem.
2. **Thread da Interface (`menu`)**
    - Permite interação do usuário via linha de comando.
    - Executa comandos como listar peers, buscar arquivos e enviar mensagens.

2.1.3 Estruturas de Dados Compartilhadas

Como múltiplas threads acessam os mesmos dados, são utilizados **mecanismos de sincronização**:

- **`peers` (Dicionário)**: Armazena os peers conhecidos no formato `{ "IP:porta": "ONLINE/OFFLINE" }`.
    - Protegido por **`peers_lock`** (mutex) para evitar condições de corrida.
- **`global_clock` (Relógio Lógico)**: Incrementado antes de enviar ou ao receber mensagens.
    - Protegido por **`clock_lock`** para garantir consistência.

2.2 Fluxo de Comunicação

2.2.1 Inicialização do Peer

1. O programa é iniciado com três parâmetros:
    - `<IP>:<porta>` (identificação do peer).
    - `<vizinhos.txt>` (arquivo com lista inicial de peers conhecidos).
    - `<diretorio_compartilhado>` (pasta com arquivos para compartilhar).
2. O peer carrega os vizinhos do arquivo e os marca como **OFFLINE**.
3. Inicia o **servidor TCP** em uma thread separada.
4. Exibe o **menu interativo** para o usuário.

2.2.2 Troca de Mensagens

Todas as mensagens seguem o formato:

<ORIGEM> <CLOCK> <TIPO> [ARGUMENTOS...]\n

Tipos de mensagens implementadas:

Tipo	Descrição	Ação ao Receber
HELLO	Usado para anunciar presença na rede.	Adiciona/atualiza peer como **ONLINE**.
GET_PEERS	Solicita a lista de peers conhecidos.	Responde com **PEER_LIST**.
PEER_LIST	Contém a lista de peers (resposta a GET_PEERS).	Atualiza lista local de peers.
BYE	Indica que o peer está saindo da rede.	Marca peer como **OFFLINE**.

Exemplo de comunicação:

1. Peer **A** envia `HELLO` para Peer **B**.
2. Peer **B** recebe a mensagem, atualiza seu relógio e marca **A** como **ONLINE**.
3. Peer **B** responde (se necessário, dependendo do tipo de mensagem).

2.3 Considerações de Projeto

- **Escalabilidade**: O sistema permite adição dinâmica de peers via mensagens `PEER_LIST`.
- **Tolerância a Falhas**: Se um peer não responde, seu status é atualizado para **OFFLINE**.
- **Thread-Safety**: Uso de **locks** para evitar corrupção de dados em acesso concorrente.
- **Extensibilidade**: O código foi estruturado para facilitar a adição de novos comandos (ex: busca de arquivos).

3. Implementação do Sistema Peer-to-Peer

Nesta seção, detalhamos os principais aspectos da implementação do sistema, incluindo:

- **Inicialização e configuração do peer**
- **Comunicação entre peers (envio/recebimento de mensagens)**
- **Gerenciamento do relógio lógico**
- **Threading e sincronização**
- **Interface do usuário e comandos disponíveis**

3.1 Inicialização do Peer

3.1.1 Parâmetros de Entrada

O programa é executado com três argumentos:

./eachare <IP:PORTA> <vizinhos.txt> <diretorio_compartilhado>

- **`<IP:PORTA>`**: Identifica o peer (ex: `127.0.0.1:5000`).
- **`<vizinhos.txt>`**: Arquivo contendo peers conhecidos inicialmente (um por linha, no formato `IP:PORTA`).
- **`<diretorio_compartilhado>`**: Pasta onde estão os arquivos compartilhados.

3.1.2 Passos de Inicialização

1. **Validação dos argumentos**:
    - Verifica se o diretório de compartilhamento existe e é acessível.
    - Analisa o arquivo `vizinhos.txt` e carrega os peers, marcando-os como `OFFLINE`.
2. **Criação do socket TCP**:
    - O peer inicia um servidor socket na porta especificada.
    - Uma thread dedicada (`server_thread`) fica escutando conexões.
3. **Início da interface de usuário**:
    - O menu interativo é exibido, permitindo comandos como listar peers e enviar mensagens.

3.2 Comunicação entre Peers

3.2.1 Formato das Mensagens

Todas as mensagens seguem o padrão:

<ORIGEM> <CLOCK> <TIPO> [ARGUMENTOS...]\n

Exemplo:

127.0.0.1:5000 5 HELLO\n

3.2.2 Envio de Mensagens

A função **`send_message()`** é responsável por:

1. Incrementar o relógio lógico antes do envio.
2. Estabelecer uma conexão TCP com o peer destino.
3. Enviar a mensagem e, se necessário, aguardar resposta.
4. Lidar com retransmissões em caso de falha (timeout ou conexão recusada).

Exemplo de envio de `HELL0`:

```
msg_clock = update_clock()  # Incrementa relógio

message = f"{server_id} {msg_clock} HELLO\n"

send_message("127.0.0.1:5001", message)
```

3.2.3 Recebimento de Mensagens

A função **`handle_client()`** processa mensagens recebidas:

- **HELLO**: Atualiza o status do remetente para `ONLINE`.
- **GET_PEERS**: Responde com uma lista de peers conhecidos (formato `PEER_LIST`).
- **BYE**: Marca o remetente como `OFFLINE`.

Exemplo de tratamento de `GET_PEERS`:

```
if msg_type == "GET_PEERS":

    lista_peers = [f"{peer}:{status}:0" for peer, status in peers.items() if peer != origem]
```

reply_msg = f"{server_id} {update_clock()} PEER_LIST {len(lista_peers)} {' '.join(lista_peers)}\n"

conn.sendall(reply_msg.encode())

3.3 Relógio Lógico (Lamport Clock)

3.3.1 Funcionamento

- Cada peer mantém um contador (`global_clock`) inicializado em `0`.
- **Antes de enviar uma mensagem**, o relógio é incrementado.
- ***Ao receber uma mensagem**, o relógio é atualizado para:*

  global_clock = max(global_clock, clock_recebido) + 1

3.3.2 Implementação

A função **`update_clock()`** garante acesso thread-safe:

def update_clock():

global global_clock

with clock_lock: # Bloqueia para evitar condições de corrida

global_clock += 1

print(f"=> Atualizando relogio para {global_clock}")

return global_clock

3.4 Threading e Sincronização

3.4.1 Thread do Servidor (`server_thread`)

- Fica em loop, aceitando conexões de outros peers.
- Para cada conexão, inicia uma **`handle_client`** em uma nova thread.

3.4.2 Thread do Menu (`menu`)

- Permite interação do usuário enquanto o servidor opera em segundo plano.

3.4.3 Mecanismos de Sincronização

- **`peers_lock`**: Protege o dicionário `peers` contra acesso concorrente.

- **`clock_lock`**: Garante atualização segura do `global_clock`.

Exemplo de uso de lock:

with peers_lock:

if peer not in peers:

peers[peer] = "OFFLINE"

3.5 Interface do Usuário

O menu oferece os seguintes comandos:

Comando	Ação
Listar peers	Mostra todos os peers conhecidos e permite enviar `HELLO`.
Obter peers	Envia `GET_PEERS` para todos, atualizando a lista de peers.
Listar arquivos	Exibe conteúdo do diretório compartilhado.
Outros	*(Não implementados nesta versão)*
Sair	Envia `BYE` para peers `ONLINE` e encerra o programa.

Exemplo de uso:

> 1 # Listar peers

Lista de peers:

[0] Voltar

[1] 127.0.0.1:5001 OFFLINE

[2] 127.0.0.1:5002 ONLINE

> 2 # Obter peers

=> Atualizando relogio para 3

Encaminhando "127.0.0.1:5000 3 GET_PEERS" para 127.0.0.1:5002

Resposta recebida: "127.0.0.1:5002 5 PEER_LIST 1 127.0.0.1:5003:ONLINE:0"

3.6 Considerações da Implementação

- **Simplicidade**: O protocolo de mensagens é baseado em texto para facilitar depuração.
- **Extensibilidade**: Novos comandos (como busca de arquivos) podem ser adicionados.
- **Robustez**: Timeouts e retentativas melhoram a tolerância a falhas.

4. Análise e Discussão

Nesta seção, avalio os principais desafios enfrentados durante a implementação, destacando as decisões de projeto, limitações encontradas e possíveis melhorias. O foco principal recai sobre a **implementação do relógio lógico**, que foi um dos componentes mais críticos do sistema.

4.1 Desafios Encontrados

4.1.1 Implementação do Relógio Lógico

Desafio:
Garantir a **consistência do relógio** em um ambiente multithread foi o principal obstáculo. Como o relógio é atualizado tanto no envio quanto no recebimento de mensagens, era necessário evitar condições de corrida (*race conditions*).

Solução adotada:

- Uso de um **lock (`clock_lock`)** para proteger o acesso ao `global_clock`.
- Sempre que o relógio é incrementado, a função `update_clock()` é chamada, garantindo que apenas uma thread modifique o valor por vez.

Problemas persistentes:

- Se múltiplas mensagens chegarem simultaneamente, pode haver um gargalo devido ao lock.

4.1.2 Comunicação TCP Confiável

Desafio:

- Peers podem falhar ou ficar inacessíveis, causando timeouts.
- O sistema precisa lidar com conexões recusadas e retransmissões.

Solução adotada:

- A função `send_message()` implementa **retentativas** (`max_retries`).
- Se um peer não responde, seu status é marcado como `OFFLINE`.

Melhoria possível:

- Adicionar um mecanismo de **heartbeat** para verificar periodicamente se um peer `OFFLINE` voltou.

4.1.3 Gerenciamento de Peers Dinâmicos

Desafio:

- A lista de peers cresce conforme novas mensagens `PEER_LIST` são recebidas.
- Pode haver inconsistência se dois peers enviarem listas diferentes ao mesmo tempo.

Solução adotada:

- Uso de **`peers_lock`** para proteger o dicionário de peers.
- Peer que envia `GET_PEERS` só atualiza sua lista após confirmar a resposta.

Limitação:

- Não há um mecanismo para remoção de peers inativos permanentemente.

4.2 Decisões de Projeto

Decisão	Justificativa
Formato textual para mensagens	Facilita depuração e é suficiente para o escopo do EP. Em sistemas reais, JSON ou binário seria mais eficiente.
Thread única para o servidor	Evita sobrecarga de múltiplas threads ouvindo na mesma porta.
Falha silenciosa em `send_message`	Se um peer não responde, ele é marcado como `OFFLINE` sem travar o programa.

4.3 Limitações do Sistema

1. **Sem autenticação ou segurança**
    - Qualquer peer pode se conectar e enviar mensagens.
    - Em um sistema real, seria necessário criptografia (ex: TLS).

2. **Relógio lógico não totalmente escalável**
    - Se muitos peers interagirem, o valor do relógio pode crescer indefinidamente.

4.4 Possíveis Melhorias

Melhoria	Benefício
Heartbeat para peers	Verificar periodicamente se peers `OFFLINE` voltaram, melhorando a descoberta.
Log de eventos	Registrar mensagens enviadas/recebidas para depuração avançada.
Tolerância a particionamento	Se a rede dividir, peers devem conseguir se reintegrar automaticamente.
Otimização do Lock	Usar `RLock` ou estruturas mais eficientes se o gargalo no relógio for crítico.

4.5 Conclusão da Análise

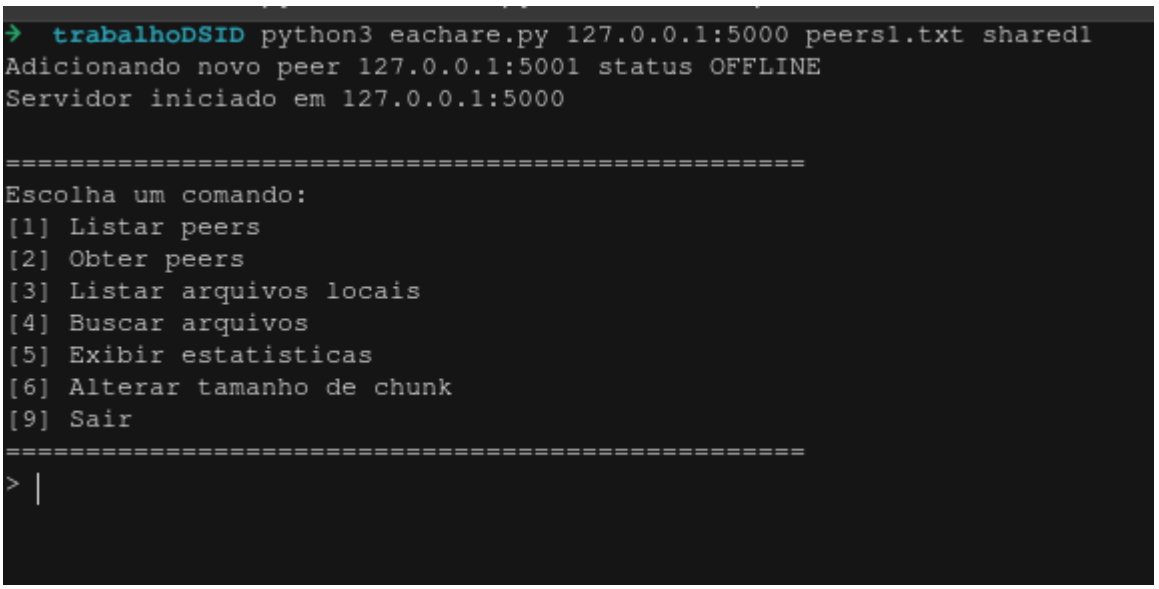
A implementação atual cumpre os requisitos básicos do EP1, com um sistema funcional de comunicação entre peers, relógio lógico e descoberta de vizinhos. O maior desafio foi garantir a **sincronização correta do relógio** em um ambiente concorrente, resolvido com locks.

5. Testes

Nesse teste vou executar o código em 3 terminais, simulando 3 máquinas.

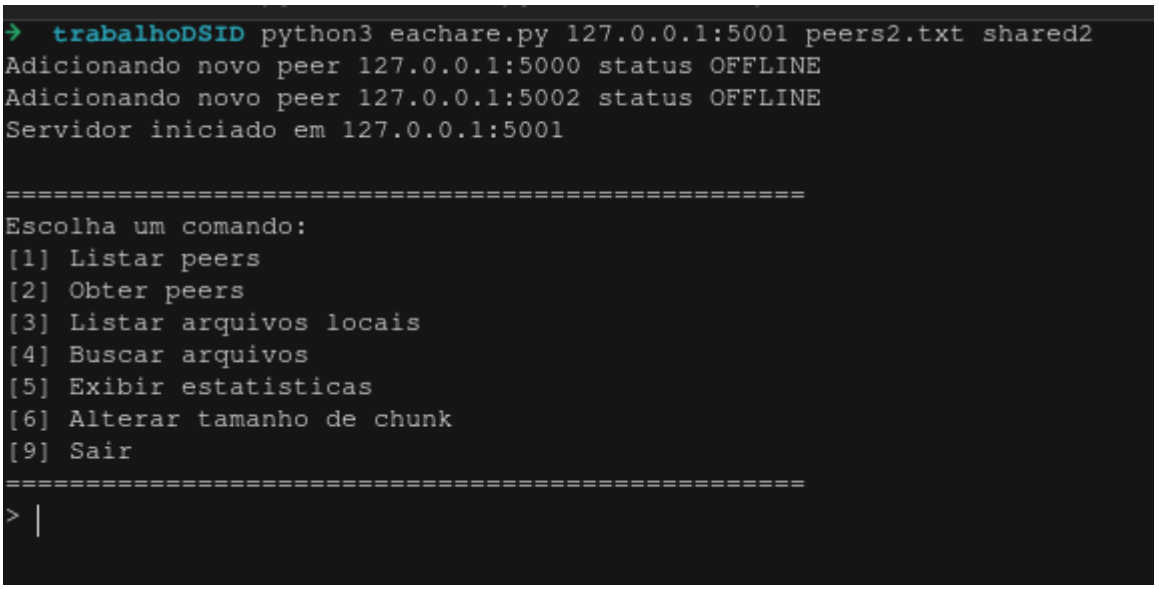
Peer 1: Conhece o Peer 2 e possui o arquivo local helloworld.txt.

Inicialização do Peer 1:



Peer 2: Conhece o Peer 1 e o Peer 3, possui o arquivo local music.mp3.

Inicialização do Peer 2:



Peer 3: Conhece o Peer 4 e possui o arquivo local objeto.java.

Inicialização do Peer 3:

```
➜  trabalhoDSID python3 eachare.py 127.0.0.1:5002 peers3.txt shared3
Adicionando novo peer 127.0.0.1:5004 status OFFLINE
Servidor iniciado em 127.0.0.1:5002

================================================
Escolha um comando:
[1] Listar peers
[2] Obter peers
[3] Listar arquivos locais
[4] Buscar arquivos
[5] Exibir estatisticas
[6] Alterar tamanho de chunk
[9] Sair
================================================
>
```

Testando comando “Listar Peers”:

Peer 1 (Enviou HELLO para Peer 2):

```
➜  trabalhoDSID python3 eachare.py 127.0.0.1:5000 peers1.txt shared1
Adicionando novo peer 127.0.0.1:5001 status OFFLINE
Servidor iniciado em 127.0.0.1:5000

================================================
Escolha um comando:
[1] Listar peers
[2] Obter peers
[3] Listar arquivos locais
[4] Buscar arquivos
[5] Exibir estatisticas
[6] Alterar tamanho de chunk
[9] Sair
================================================
> 1

Lista de peers:
[0] Voltar para o menu anterior
[1] 127.0.0.1:5001 OFFLINE
> 1
=> Atualizando relogio para 1
Encaminhando mensagem "127.0.0.1:5000 1 HELLO" para 127.0.0.1:5001
Atualizando peer 127.0.0.1:5001 status ONLINE

Lista de peers:
[0] Voltar para o menu anterior
[1] 127.0.0.1:5001 ONLINE
>
```

Peer 2 (Recebeu HELLO do Peer 1 e enviou HELLO para Peer 3):

```
==================================================
> Mensagem recebida: "127.0.0.1:5000 1 HELLO"
=> Atualizando relogio para 1
Atualizando peer 127.0.0.1:5000 status ONLINE
1

Lista de peers:
[0] Voltar para o menu anterior
[1] 127.0.0.1:5000 ONLINE
[2] 127.0.0.1:5002 OFFLINE
> 2
=> Atualizando relogio para 2
Encaminhando mensagem "127.0.0.1:5001 2 HELLO" para 127.0.0.1:5002
Atualizando peer 127.0.0.1:5002 status ONLINE

Lista de peers:
[0] Voltar para o menu anterior
[1] 127.0.0.1:5000 ONLINE
[2] 127.0.0.1:5002 ONLINE
>
```

Peer 3 (Recebeu HELLO do PEER 2 e enviou HELLO para peer 4 que estava inativo):

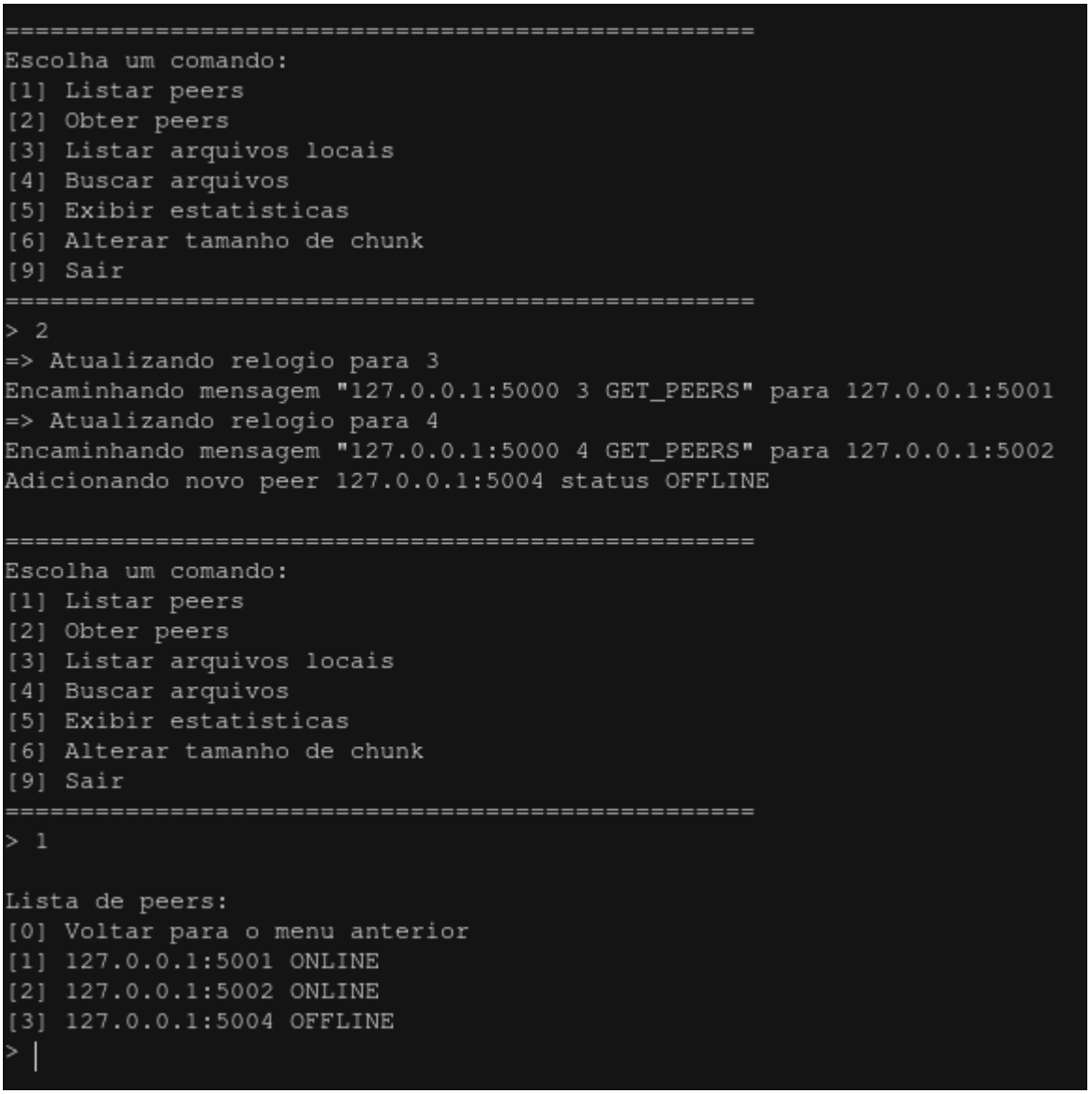
```
> Mensagem recebida: "127.0.0.1:5001 2 HELLO"
=> Atualizando relogio para 1
Atualizando peer 127.0.0.1:5001 status ONLINE
1

Lista de peers:
[0] Voltar para o menu anterior
[1] 127.0.0.1:5004 OFFLINE
[2] 127.0.0.1:5001 ONLINE
> 1
=> Atualizando relogio para 2
Conexão recusada por 127.0.0.1:5004

Lista de peers:
[0] Voltar para o menu anterior
[1] 127.0.0.1:5004 OFFLINE
[2] 127.0.0.1:5001 ONLINE
>
```

Testando comando “Obter Peers”:

Vou executar esse teste apenas com o Peer 1, ele inicialmente apenas conhece o Peer 2, e o Peer 2 conhece o Peer 3, então ao obter peers a partir do Peer 1, ele deve passar a conhecer o Peer 3 graças ao Peer 2.

```
Escolha um comando:
[1] Listar peers
[2] Obter peers
[3] Listar arquivos locais
[4] Buscar arquivos
[5] Exibir estatisticas
[6] Alterar tamanho de chunk
[9] Sair
==================================================
> 1

Lista de peers:
[0] Voltar para o menu anterior
[1] 127.0.0.1:5001 ONLINE
> 0

==================================================
Escolha um comando:
[1] Listar peers
[2] Obter peers
[3] Listar arquivos locais
[4] Buscar arquivos
[5] Exibir estatisticas
[6] Alterar tamanho de chunk
[9] Sair
==================================================
> 2
=> Atualizando relogio para 2
Encaminhando mensagem "127.0.0.1:5000 2 GET_PEERS" para 127.0.0.1:5001
Adicionando novo peer 127.0.0.1:5002 status ONLINE

==================================================
Escolha um comando:
[1] Listar peers
[2] Obter peers
[3] Listar arquivos locais
[4] Buscar arquivos
[5] Exibir estatisticas
[6] Alterar tamanho de chunk
[9] Sair
==================================================
> 1

Lista de peers:
[0] Voltar para o menu anterior
[1] 127.0.0.1:5001 ONLINE
[2] 127.0.0.1:5002 ONLINE
> |
```

Como esperado, o Peer 1 passou a conhecer o Peer 3.
Vou obter peers de novo, agora, como o Peer 3 conhece o Peer 4, o Peer 1 deve conhecer também o Peer 4 após o comando.



Como esperado, o Peer 1 agora conhece o Peer 4.

Testando comando “Listar arquivos locais”:

Peer 1:

```
==================================================
Escolha um comando:
[1] Listar peers
[2] Obter peers
[3] Listar arquivos locais
[4] Buscar arquivos
[5] Exibir estatisticas
[6] Alterar tamanho de chunk
[9] Sair
==================================================
> 3

Arquivos no diretório compartilhado:
helloworld.txt

==================================================
```

Peer 2:

```
==================================================
Escolha um comando:
[1] Listar peers
[2] Obter peers
[3] Listar arquivos locais
[4] Buscar arquivos
[5] Exibir estatisticas
[6] Alterar tamanho de chunk
[9] Sair
==================================================
> 3

Arquivos no diretório compartilhado:
music.mp3

==================================================
```

Peer 3:

```
==================================================
Escolha um comando:
[1] Listar peers
[2] Obter peers
[3] Listar arquivos locais
[4] Buscar arquivos
[5] Exibir estatisticas
[6] Alterar tamanho de chunk
[9] Sair
==================================================
> 3

Arquivos no diretório compartilhado:
objeto.java

==================================================
```

Testando comando “Sair”:

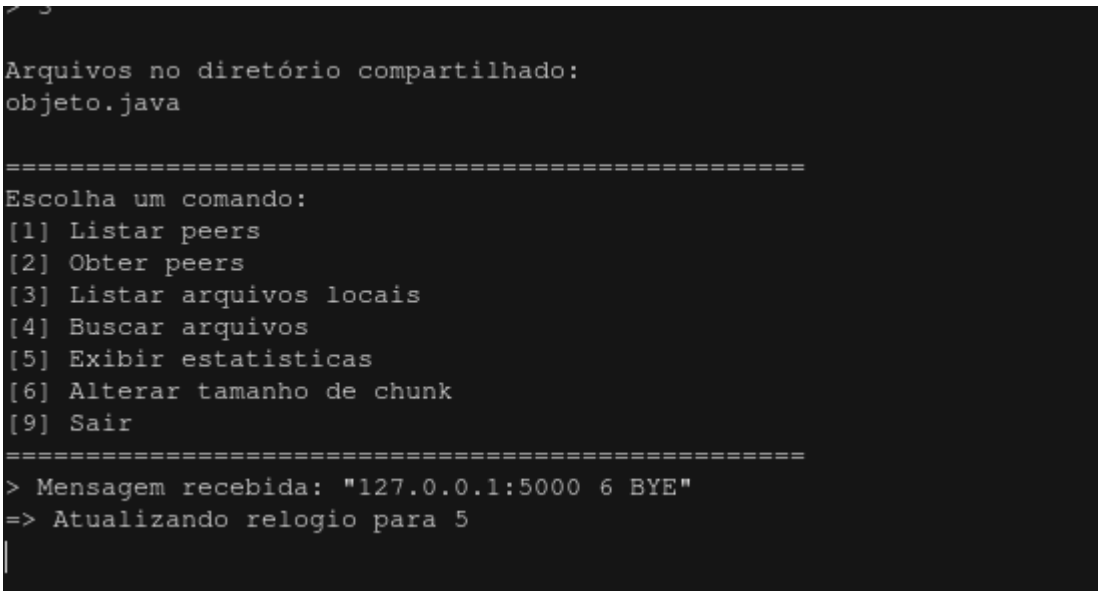
Nesse teste vou executar o comando Sair apenas com o Peer 1, o Peer 1 deve enviar uma mensagem de “BYE” para o Peer 2 e Peer 3:

```
================================================
Escolha um comando:
[1] Listar peers
[2] Obter peers
[3] Listar arquivos locais
[4] Buscar arquivos
[5] Exibir estatisticas
[6] Alterar tamanho de chunk
[9] Sair
================================================
> 9

Saindo...
=> Atualizando relogio para 5
Encaminhando mensagem "127.0.0.1:5000 5 BYE" para 127.0.0.1:5001
=> Atualizando relogio para 6
Encaminhando mensagem "127.0.0.1:5000 6 BYE" para 127.0.0.1:5002
➜  trabalhoDSID
```

Mensagem recebida pelo Peer 2:

```
Arquivos no diretório compartilhado:
music.mp3

================================================
Escolha um comando:
[1] Listar peers
[2] Obter peers
[3] Listar arquivos locais
[4] Buscar arquivos
[5] Exibir estatisticas
[6] Alterar tamanho de chunk
[9] Sair
================================================
> Mensagem recebida: "127.0.0.1:5000 5 BYE"
=> Atualizando relogio para 7
Atualizando peer 127.0.0.1:5000 status OFFLINE
```

Mensagem recebida pelo Peer 3:



6. Conclusão

O desenvolvimento deste sistema peer-to-peer (P2P) representou um desafio significativo no aprendizado de conceitos fundamentais de **redes distribuídas, concorrência e sincronização**. A implementação cumpriu os requisitos básicos do EP1, permitindo a comunicação entre peers, manutenção de um relógio lógico e descoberta dinâmica de nós na rede.

6.1 Principais Conquistas

1. **Protocolo de Comunicação Funcional**
    - O sistema estabeleceu um padrão de mensagens baseado em texto, com suporte a `HELLO`, `GET_PEERS` e `BYE`, permitindo interação básica entre os nós.
2. **Relógio Lógico Operacional**
    - A implementação do relógio garantiu uma ordenação parcial de eventos, resolvendo condições de corrida com o uso de **locks**.
3. **Arquitetura Concorrente Eficiente**
    - A separação entre **thread do servidor** (que escuta conexões) e **thread do menu** (que processa comandos) permitiu um sistema responsivo e não-bloqueante.
4. **Extensibilidade**
    - A estrutura adotada permite a fácil adição de novos comandos (como busca de arquivos) em fases futuras do projeto.

6.2 Lições Aprendidas

- **Concorrência é complexa, mas gerenciável**: O uso de **locks** (`threading.Lock`) foi essencial para evitar inconsistências no relógio e na lista de peers.
- **O paradigma híbrido (imperativo + OOP) foi adequado**, mas em sistemas maiores, uma abordagem mais orientada a objetos seria preferível.
- **TCP exige tratamento robusto de falhas**: O sistema precisou lidar com timeouts e peers offline, mostrando a importância de **mecanismos de "re-tentativa"**.

6.3 Trabalhos Futuros

Para expandir o sistema, as seguintes melhorias são recomendadas:

1. **Implementar busca de arquivos** usando *flooding* ou tabelas hash distribuídas (DHT).
2. **Adicionar segurança**, como autenticação básica ou criptografia nas mensagens.
3. **Substituir threads por async/await** (`asyncio`) para melhor desempenho em ambientes com muitos peers.
4. **Melhorar o protocolo**, adotando JSON ou binário para mensagens mais estruturadas.

6.4 Considerações Finais

Este projeto demonstrou que mesmo um sistema P2P básico exige cuidados com **sincronização, concorrência e tolerância a falhas**. A solução desenvolvida serve como base para implementações mais complexas, reforçando a importância de um design modular e bem estruturado.

O código-fonte e a abordagem adotada estão prontos para evoluir, seja para atender aos próximos EPs ou para explorar novos recursos em redes distribuídas.