ACH2147 – Desenvolvimento de Sistemas de Informação Distribuídos

1° semestre de 2025

# Exercício Programa: parte 1

Felipe Freire Silva

## Sumário

# 1 Objetivos

O objetivo deste exercício programa é implementar um sistema de compartilhamento de arquivos peer-to-peer simplificado chamado EACHare. Cada peer participante da rede disponibiliza um conjunto de arquivos e mantém uma lista de outros peers conhecidos. Qualquer peer da rede pode perguntar aos seus peers conhecidos quais arquivos eles possuem e selecionar um dos arquivos para fazer o download.

# 2 Decisões de projeto

Nesta seção, são apresentadas as principais decisões de projeto tomadas para a implementação da Parte 1 do EACHare. O foco desta etapa foi o gerenciamento de peers em uma rede P2P, utilizando troca de mensagens, manutenção de um relógio lógico distribuído e controle do estado de conectividade entre os peers.

As escolhas feitas buscaram modularidade, garantindo que o código fosse funcional e extensível para as próximas fases do EP e de fácil manutenção. Foram aplicadas boas práticas de programação, como separação de responsabilidades em

diferentes módulos, uso de threads para lidar com concorrência e um modelo de comunicação baseado em sockets TCP.

## 2.1 Dificuldades encontradas

A comunicação entre peers com sockets TCP foi a maior dificuldade desta primeira parte, uma vez que eu nunca havia utilizado tal recurso na prática, portanto foi gasto um tempo adicional para compreender o funcionamento e a implementação de sockets no código, além de testes manuais para facilitar a compreensão.

Portanto, a dificuldade foi a troca de mensagens entre peers, que exigiu o uso de sockets TCP, isso gerou a necessidade de lidar com conexões simultâneas, abertura e fechamento de sockets, possíveis exceções de rede (como timeouts e recusas de conexão), e a formatação correta das mensagens. Foi necessário garantir que os peers pudessem se comunicar entre si de forma confiável e sem interromper a execução do menu interativo, o que exigiu a criação de threads.

## 2.2 Paradigma de programação

O paradigma de programação adotado neste projeto foi a programação orientada a objetos. O motivo da escolha foi a necessidade de organizar diferentes responsabilidades do sistema de forma escalável, de modo que facilite a manutenção e a extensão nas próximas partes do exercício programa.



A arquitetura do sistema foi dividida em diversas classes, cada uma com responsabilidades bem definidas:

- Peer (peer.py): é a classe principal do sistema, responsável por representar um peer ativo da rede. Ela encapsula atributos como endereço, porta, vizinhos conhecidos, socket de comunicação e instância do relógio lógico. Além disso, ela centraliza a lógica de inicialização, escuta de conexões, tratamento de mensagens recebidas e interação com o restante da rede.
- Mensagem (mensagem.py): ela é responsável por construir, codificar e decodificar mensagens, além de permitir a visualização no terminal com a função auxiliar de exibir envio. O uso de uma classe específica para mensagens permitiu centralizar a lógica de formatação e análise das strings, evitando repetição de código.
- Relogio (relogio.py): implementa o relógio lógico de Lamport como uma entidade independente, com métodos para incrementar o clock no envio e no recebimento de mensagens. A separação em uma classe própria ajuda a manter a coerência da atualização do clock em todo o sistema, além de facilitar a depuração com mensagens no terminal.

Além das classes, o projeto também utiliza funções organizadas no arquivo menu.py, que compõem a interface de interação do usuário. Embora menu.py não use uma classe própria, sua separação permite manter a interface do usuário desacoplada da lógica de rede e da lógica de estado do peer.

Por fim, o main.py contém apenas o ponto de entrada do programa. Ele é responsável por interpretar os argumentos da linha de comando, instanciar o peer e iniciar o menu interativo. Isso resulta em encapsular a lógica de controle no ponto de entrada e delegar as responsabilidades específicas às classes adequadas.

## 2.3 Threads

A divisão do programa em múltiplas threads foi feita para garantir o funcionamento simultâneo do menu interativo com o usuário e da comunicação entre peers na rede.

Na implementação, essa divisão foi feita:

- A thread principal do programa é responsável por executar o menu interativo. Nela, o usuário pode escolher comandos como listar peers, obter peers, listar arquivos locais ou sair do programa. Essa thread controla o fluxo da aplicação do ponto de vista do usuário.
- Ao iniciar o peer, o programa cria uma segunda thread dedicada para escutar conexões TCP de entrada. Essa thread fica em execução contínua, aguardando conexões iniciadas por outros peers que desejam enviar mensagens.



- Sempre que essa thread de escuta aceita uma nova conexão, uma nova thread individual é criada especificamente para tratar aquela conexão. Essa thread lida com a leitura da mensagem recebida, atualização do relógio lógico, verificação do tipo de mensagem e execução da ação correspondente (como atualizar status ou responder com lista de peers).

Com essa divisão, o sistema consegue:

- Manter o menu do usuário sempre responsivo
- Receber mensagens de múltiplos peers simultaneamente
- Processar cada mensagem de forma isolada, sem risco de bloqueio

Além disso, todas as threads criadas para escutar ou tratar conexões foram configuradas como threads auxiliares, o que significa que elas não impedem o encerramento do programa quando o usuário decide sair.

## 2.4 Bloqueantes

Para o envio e recebimento de dados entre os peers, foi adotada a abordagem com operações bloqueantes. Essa decisão foi tomada por ser a mais simples de implementar nesta primeira fase do projeto e por se adequar bem ao modelo baseado em threads concorrentes, utilizado para tratar conexões e mensagens.

Quando o programa tenta enviar ou receber dados por um socket, a execução da thread que fez essa chamada é interrompida até que a operação seja concluída. Por exemplo:

- Se o peer tentar se conectar a outro peer e o destino não estiver disponível, a thread ficará bloqueada por um tempo até a tentativa falhar.
- Se o peer estiver aguardando o recebimento de uma mensagem, a thread esperará até que a mensagem chegue.

No contexto deste projeto, o uso de chamadas bloqueantes não prejudica a responsividade do sistema, porque todas as interações com a rede são feitas em threads separadas da thread principal, que executa o menu do usuário. Assim, mesmo que uma conexão demore ou falhe, a interface com o usuário permanece funcionando normalmente.

Essa abordagem também simplificou o tratamento de falhas de conexão. Sempre que uma tentativa de conexão ou envio de mensagem não foi bem-sucedida, foi possível detectar imediatamente a exceção gerada e atualizar o status do peer correspondente para OFFLINE, mantendo a integridade da rede.

Por fim, utilizar operações bloqueantes facilitou o raciocínio sobre a ordem das ações e o controle do relógio lógico, que pôde ser atualizado com antes e após cada operação de rede, sem a necessidade de controle assíncrono.



## 2.5 Estruturas de dados

A escolha das estruturas de dados foi feita com base na eficiência necessária para implementar as funcionalidades da Parte 1 do projeto EACHare, o qual envolve gerenciar peers conhecidos, troca de mensagens e o controle do relógio lógico.

**Dicionário** (dict) foi usado na classe Peer, para armazenar os peers conhecidos e seus respectivos status (ONLINE ou OFFLINE). Ele permite o acesso rápido a qualquer peer conhecido por meio da sua chave, que é o endereço no formato ip:porta. O valor armazenado é o status atual do peer. A estrutura de dicionário oferece complexidade de tempo constante para operações de busca e atualização, o que é ideal para verificar rapidamente se um peer já é conhecido, atualizar seu status ou adicionar novos peers conforme chegam mensagens do tipo HELLO, BYE ou PEER_LIST.

**Listas** (list) foi usado para montar a lista de argumentos das mensagens (Mensagem.argumentos), para armazenar e iterar sobre os arquivos locais no diretório compartilhado, para apresentar a lista de peers no menu interativo e selecionar qual peer deve receber uma mensagem e para processar a lista de peers recebida em uma resposta PEER_LIST. As listas permitem o

armazenamento ordenado e a manipulação sequencial de dados, com iteração fácil e indexação direta. A natureza linear e flexível das listas torna essa estrutura ideal para organizar itens que precisam ser exibidos ao usuário ou transmitidos em sequência dentro das mensagens.

## 2.6 Testes

O processo de testes do projeto foi dividido em duas etapas principais: inicialmente, os testes foram realizados de forma manual, com o objetivo de entender o funcionamento dos sockets na prática e verificar o comportamento das mensagens entre peers em tempo real. Em seguida, foram implementados testes automatizados para facilitar a validação do sistema e garantir a reprodutibilidade dos resultados.

Durante o desenvolvimento inicial, os testes foram conduzidos manualmente, executando múltiplas instâncias do programa em diferentes terminais com configurações distintas de endereço e arquivo de vizinhos. Essa abordagem permitiu verificar visualmente:

- A troca de mensagens entre os peers (HELLO, GET_PEERS)
- A atualização do relógio lógico a cada envio ou recebimento
- A consistência da lista de peers conhecida por cada instância
- O funcionamento da interface do menu interativo

Esses testes ajudaram especialmente a entender o funcionamento do modelo de comunicação com sockets TCP, o comportamento de conexões bloqueantes e como threads paralelas podem receber e tratar mensagens ao mesmo tempo em que o usuário interage com o programa.



Após a validação manual, foram desenvolvidos scripts automatizados de teste. O script realiza automaticamente o seguinte:

- Criação de múltiplos peers simultaneamente
    - Simula a execução de três instâncias do programa em subprocessos
    - Usa arquivos de vizinhos e diretório compartilhado predefinidos
- Envio de mensagens entre os peers
    - Envia HELLO de um peer para outro, para testar a atualização do status
    - Envia GET_PEERS e verifica se a resposta PEER_LIST está formatada corretamente e contém os peers esperados
- Verificação do diretório compartilhado
    - Confirma se o diretório compartilhado é acessível

    - Verifica se ele contém arquivos, garantindo que a funcionalidade de listagem local está operando
- Recebimento e interpretação de mensagens
    - Observa se os peers reagem corretamente a mensagens recebidas e atualizam seus estados internos conforme esperado
- Geração de relatório
    - Todos os testes executados pelo script são registrados em um arquivo contendo o resultado de cada etapa (SUCESSO ou FALHA)

# 3 Implementação

Nesta seção, apresentamos um resumo das principais partes da implementação do projeto EACHare – Parte 1. A arquitetura do sistema foi desenvolvida com foco em modularidade, clareza e facilidade de manutenção, permitindo que as responsabilidades do sistema fossem distribuídas de forma lógica entre diferentes arquivos e classes.

A seguir, serão descritos os principais componentes do sistema, abordando a forma como cada um contribui para o funcionamento geral da aplicação. Cada subseção resume a finalidade de uma parte do código, que se encontra devidamente comentado nos arquivos entregues.



## 3.1 Mensagem

Classe que define o formato padrão das mensagens trocadas entre peers. Encapsula os campos de origem, relógio, tipo e argumentos. Oferece métodos para codificar uma mensagem para envio e decodificar uma mensagem recebida. Garante consistência na estrutura das mensagens e facilita o parsing centralizado em toda a aplicação.

A comunicação entre os peers foi implementada com base em um protocolo textual simples, definido pela classe Mensagem. Essa classe é responsável por construir, codificar e decodificar todas as mensagens trocadas entre peers, no formato especificado pelo enunciado: <origem> <clock> <tipo> [argumentos].

## 3.2 Relógio

Classe responsável pelo relógio lógico de Lamport. Mantém o valor atual do clock

e atualiza corretamente em eventos de envio e recebimento. Permite ao peer registrar causalidade entre mensagens distribuídas sem depender de sincronização física de tempo. Fornece mensagens no terminal sempre que o clock é incrementado.

## 3.3 Peer

Classe principal que representa o peer ativo na rede. Gerencia o endereço, a porta, a lista de peers conhecidos, o socket TCP e o diretório de arquivos compartilhados.
Responsável por iniciar a escuta de conexões e tratar cada mensagem recebida em uma thread separada. Interage com o relógio lógico e executa ações com base no tipo da mensagem (HELLO, GET_PEERS, BYE, etc). Também oferece suporte ao envio de mensagens para outros peers e à atualização do status de conectividade.

## 3.4 Comando [1]

Apresenta ao usuário todos os peers conhecidos pelo peer local, indicando seus respectivos status (ONLINE ou OFFLINE). Permite escolher um peer da lista para enviar uma mensagem HELLO, usada para verificar a conectividade. Se a mensagem for entregue com sucesso, o status do peer é atualizado para ONLINE. Se houver falha de conexão, o status é alterado para OFFLINE.



## 3.5 Comando [2]

Envia a mensagem GET_PEERS para todos os peers atualmente conhecidos. Cada peer que recebe essa mensagem responde com uma PEER_LIST, contendo os peers que ele conhece. O peer local então analisa a resposta e adiciona novos peers à sua lista, ou atualiza o status dos já existentes. Esse comando permite expandir a rede de contatos de forma dinâmica e colaborativa.

## 3.6 Comando [3]

Lista todos os arquivos presentes no diretório compartilhado configurado pelo peer local. A funcionalidade é local, ou seja, não depende da rede nem da troca de mensagens. Permite ao usuário visualizar os arquivos disponíveis para possível compartilhamento futuro. Exibe uma mensagem de erro caso o diretório não exista ou não seja acessível.

## 3.7 Comando [9]

Encerra a execução do peer de forma controlada. Antes de finalizar, envia a mensagem BYE para todos os peers com status ONLINE, informando que o peer local está saindo da rede. Os peers que recebem o BYE atualizam o status do remetente para OFFLINE. Após o envio, o programa encerra todas as operações e libera os recursos utilizados.

# 4 Conclusão

A Parte 1 do exercício programa EACHare implementou a base para um sistema distribuído de compartilhamento de arquivos. Todas as funcionalidades relacionadas ao gerenciamento de peers, troca de mensagens e controle de relógio lógico foram desenvolvidas conforme especificado.