UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ARTES, CIÊNCIAS E HUMANIDADES

Gabriel Silva Pereira - 13833125

Jonathan de Ramos - 10857082

Relatório do EP1

Desenvolvimento de Sistemas de Informação Distribuídos

Professor Renan Cerqueira

São Paulo

2025

Relatório do EP1

Trabalho apresentado à Escola de Artes, Ciências e Humanidades, à disciplina de Desenvolvimento de Sistemas de Informação Distribuídos.

Professor orientador: Renan Cerqueira Afonso Alves

São Paulo

2025

Sumário

Introdução

Este relatório descreve o desenvolvimento de um sistema de compartilhamento de arquivos peer-to-peer simplificado, de acordo com as especificações dadas pelo enunciado do exercício- programa (EP), chamado *EACHare*. O objetivo principal do trabalho foi permitir que cada *peer* (ou nó da rede) disponibilizasse um conjunto de arquivos e mantivesse uma lista de outros *peers* conhecidos, por meio da implementação de algumas funções. Através deste sistema, qualquer *peer* pode consultar seus conhecidos para descobrir quais arquivos eles possuem e, em seguida, realizar o *download* de um arquivo específico.

O relatório foi feito para contemplar as principais decisões de projeto, no que envolve tópicos como paradigma de programação escolhido, estruturas de dados utilizadas, forma de organização do código e afins. Não apenas isso, também irá relatar nossas maiores dificuldades encontradas no desenvolvimento do projeto, bem como considerações acerca de testes e demais informações relevantes para completo entendimento do nosso trabalho.

Desenvolvimento do projeto

Para iniciarmos o desenvolvimento do código do EACHare, precisávamos escolher a linguagem de programação que melhor atenderia às necessidades do nosso sistema peer to peer. Após ponderações, escolhemos utilizar a linguagem Python, pois ela oferece suporte nativo para criação de sockets e threads, além de facilitar a escrita de um código claro e conciso. Falando em threads, a divisão do programa dessa forma foi pensada para simplificar a lógica de concorrência: isso porque uma thread se encarrega de aceitar e tratar conexões, enquanto outra cuida do menu de interação com o usuário. O paradigma de programação escolhido foi o procedural, visto que essa primeira parte do exercício-programa não exige uma hierarquia de classes nem polimorfismo, já que grande parte do projeto se baseia em ler parâmetros, gerenciar *peers* e enviar/receber mensagens.

Divisão do Programa em *Threads*

O programa cria uma thread principal responsável por:

1. Iniciar o servidor TCP e colocá-lo em modo de escuta.

2. Executar o loop do menu, onde o usuário pode escolher opções como listar *peers*, obter *peers*, listar arquivos, etc.

Em paralelo, há a thread do servidor (server_thread()), que:

- Fica em loop chamando server_socket.accept() e, para cada conexão que chega, inicializa uma nova *thread* (handle_connection()), encarregada de processar a requisição até o final.

- Essa nova *thread* lerá a mensagem, atualizará o relógio e tomará ações específicas dependendo do tipo de mensagem (por exemplo, HELLO, GET_PEERS, PEER_LIST, BYE).

Essa arquitetura em que cada conexão recebida gera uma *thread* de serviço é conhecida como *thread-per-connection* ou *thread-per-request*

Nas trocas de dados, utilizamos operações bloqueantes de envio e recebimento (em Python, o uso de socket.recv() e socket.sendall() já é bloqueante por padrão), por serem mais diretas de implementar e controlarem melhor a sincronização na comunicação entre peers. Antes de cada envio de mensagem e a cada recepção, o relógio local é incrementado e seu novo valor é exibido no console, o que facilita acompanhar o comportamento distribuído do sistema em tempo real. Em paralelo, o gerenciamento do estado de cada peer fica registrado, sendo atualizado tanto ao detectar sucesso ou falha no envio de mensagens quanto no recebimento dos tipos HELLO ou BYE, o que mantém as informações de rede consistentes. Para que o peer não bloqueie o processamento global enquanto o usuário interage com o menu, optamos por utilizar threads: há uma thread dedicada ao servidor TCP que aceita conexões e, para cada conexão estabelecida, outra thread é criada para processar a requisição.

Desse modo, a implementação do *EACHare* seguiu uma abordagem incremental, focada em possibilitar a comunicação entre os *peers* por meio de *sockets* TCP, respeitando o formato de mensagem definido, mantendo um relógio lógico simples e gerenciando a lista de *peers* conhecidos. Seguindo o enunciado do projeto, nosso código recebe o endereço e a porta do peer, bem como o arquivo de parâmetros, todos passados pela linha de comando. Tendo a lista de vizinhos e o diretório de arquivos compartilhados lidos a partir do arquivo de parâmetros, isso garante flexibilidade ao rodar vários *peers* numa mesma máquina ou em máquinas distintas.

A fim de manter as informações sobre peers, optamos por utilizar um dicionário Python em que as chaves são strings no formato “ip:porta” e os valores são outros dicionários contendo campos como “address” (para o IP), “port” (para a porta) e “status” (com valores “ONLINE” ou “OFFLINE”). Essa escolha se justifica pela simplicidade e eficiência, pois o dicionário permite rápidas inserções e buscas (geralmente em tempo constante), o que é muito conveniente para operações de rede em que o programa pode receber ou enviar mensagens a qualquer momento tal como nosso sistema peer to peer.

Como as atualizações nos dados de peers acontecem em threads concorrentes, foi necessário adotar locks para evitar condições de corrida e garantir a consistência tanto do dicionário quanto do nosso relógio local. Por exemplo, sempre que um peer muda de “OFFLINE” para “ONLINE” ou vice-versa, ou quando o relógio é incrementado ao enviar/receber mensagens, realizamos o bloqueio adequado antes de efetuar as alterações. Assim, independentemente do número de threads em operação, não ocorre sobreposição de acessos que possa corromper o estado global.

Dificuldades Enfrentadas

Durante o desenvolvimento do EACHare, a concorrência e a segurança de dados foram pontos particularmente delicados. Como manipulamos variáveis globais – especialmente o relógio lógico e o dicionário de peers – em um contexto multi-threaded, precisávamos evitar condições de corrida na hora de atualizar o valor do relógio ou alterar o estado de um peer. Para isso, empregamos locks (fornecidos pelo módulo threading do Python), o que garantiu que incrementos no relógio e modificações no dicionário não ocorressem simultaneamente em diferentes threads. Outro aspecto sensível foi o tratamento de exceções na comunicação via socket: falhas de conexão, timeouts ou encerramentos inesperados pelo lado remoto poderiam deixar o sistema em um estado inconsistente, então isolamos a lógica de envio e recebimento de dados em funções que lidam cuidadosamente com erros, retornando valores de forma coerente quando algo não sai como planejado.

Além disso, tivemos de pensar na sincronização da saída no console, já que múltiplas threads imprimindo mensagens ao mesmo tempo podem resultar em trechos de texto intercalados. Embora tenhamos optado por uma solução simples que não impede totalmente esse entrelaçamento, garantimos que cada mensagem apareça de forma clara para o usuário final. Para superar todas essas dificuldades, estudamos a fundo a biblioteca de threading do Python e aprofundamos nosso entendimento do funcionamento de sistemas peer to peer, resultando em uma arquitetura robusta e escalável que lida bem com o ambiente distribuído.

Testes Realizados

Para garantir que cada funcionalidade do EACHare atendesse às especificações, realizamos uma bateria de testes **manuais e incrementais** dentro do Visual Studio Code. Em nosso diretório de projeto, criamos várias pastas, cada uma contendo um arquivo de texto diferente, simulando os arquivos compartilhados por cada peer. Assim, pudemos verificar tanto a correta listagem de arquivos locais como a capacidade dos peers de se comunicarem em rede. A seguir, destacamos os principais cenários de teste:

1. **Teste de Inicialização**
   Iniciamos o programa com parâmetros válidos – por exemplo, python eachare.py 127.0.0.1:9001 vizinhos.txt ./pastaPeer1 – e verificamos se o menu principal era exibido corretamente. Também confirmamos que o servidor TCP subia sem erros, que a mensagem “Adicionando novo peer” era mostrada para cada peer listado em vizinhos.txt e que o programa identificava adequadamente quaisquer parâmetros inválidos, como um diretório inexistente ou porta fora do intervalo.

2. **Teste de Comunicação HELLO**
   Executamos ao menos dois peers em terminais diferentes. Em um deles, escolhemos a opção de “Listar peers” e enviamos mensagens do tipo HELLO para o outro, observando o incremento do relógio tanto no envio quanto no recebimento da

mensagem, refletido no console. Também checamos a atualização do status do peer, que passava de OFFLINE para ONLINE ao estabelecer contato bem-sucedido. Esse procedimento também foi útil para detectar se o destinatário não estava realmente em execução, pois, nesse caso, o status permanecia OFFLINE após tentar o envio.

3. **Teste de Obter Peers (GET_PEERS)**
   Verificamos o encadeamento da descoberta de peers ao enviar GET_PEERS para um vizinho que, por sua vez, retornava uma lista contendo outros peers. Esse teste validou se o receptor conseguia inserir essas novas informações no dicionário local, mantendo a rede atualizada à medida que diferentes peers eram descobertos dinamicamente.

4. **Teste de Listagem de Arquivos**
   Cada pasta de peer continha um ou mais arquivos de teste (por exemplo, “arquivo1.txt”). Ao selecionarmos a opção de listar arquivos locais, confirmamos se o sistema exibia corretamente o nome de cada arquivo ou, em caso de pasta vazia, se a mensagem “Diretório vazio” aparece. Esse passo permitiu verificar a coerência entre o peer e a pasta real que ele gerenciava.

5. **Teste de Encerramento (Sair)**
   Para testar o envio de **BYE**, selecionamos a opção de sair no menu. Observamos se a aplicação percorria corretamente todos os peers em estado ONLINE e enviava a mensagem de desligamento. No outro terminal, o peer receptor acusava a chegada do BYE e alterava o status do emissor para OFFLINE. Dessa forma, confirmamos que o procedimento de saída não deixava peers “presos” em estados indevidos.

No geral, esses testes práticos, que abrangeram desde a inicialização até o encerramento de um peer, nos deram confiança de que o funcionamento básico do protocolo estava de acordo com o enunciado. Além disso, pudemos detectar e corrigir pequenas inconsistências ao longo do processo, garantindo uma maior robustez para futuras expansões do EACHare

Execução

Para executar o programa, abra um terminal e navegue até o diretório onde o arquivo eachare.py está localizado. Certifique-se de ter criado um arquivo de texto, por exemplo, vizinhos.txt, contendo em cada linha os endereços e portas dos peers no formato <endereco>:<porta> (por exemplo, "127.0.0.1:9091", "127.0.0.1:9092", etc.) e também de ter um diretório (por exemplo, compartilhados/) com os arquivos (por exemplo o proprio enunciado do EP1 e outro arquivo txt) que serão compartilhados. Em seguida, inicie o programa para o primeiro peer com o comando:

```
python3 eachare.py 127.0.0.1:9090 vizinhos.txt compartilhados/
```

Para testar a comunicação e o gerenciamento dos peers, **abra novos terminais e execute instâncias adicionais do programa utilizando portas diferentes (Certifique-se que as portas utilizadas estão disponíveis, em nosso teste utilizamos as portas 9090, 9091 e 9092)**, de acordo com os peers listados no arquivo vizinhos.txt. Por exemplo, para iniciar outros peers, você pode usar os comandos:

```
python3 eachare.py 127.0.0.1:9091 vizinhos.txt compartilhados/
```

```
python3 eachare.py 127.0.0.1:9092 vizinhos.txt compartilhados/
```

Dessa forma, cada instância ficará escutando na sua porta correspondente e poderá se comunicar com os demais peers via sockets TCP.

Conclusão

A implementação do *EACHare* mostrou-se funcional para a proposta da primeira parte, viabilizando um sistema de compartilhamento de arquivos em que cada *peer* atua tanto como cliente quanto como servidor (*peer-to-peer*) de maneira simplificada. Além de reforçar o conhecimento sobre *threads* e gerenciamento de *locks* em Python, o trabalho evidenciou a importância de seguir corretamente um protocolo definido e de cuidar da consistência dos dados em um ambiente concorrente, algo muito importante quando tratamos de sistemas distribuídos.