UNIVERSIDADE FEDERAL DE ITAJUBÁ

SISTEMAS DE INFORMAÇÃO

Uma Solução Descentralizada para o Compartilhamento Seguro de Mídias em uma Agência de Marketing Colaborativa

2020004243 - MARCELO MAGALHÃES SILVA
2018003703 - HENRIQUE CASTRO OLIVEIRA
2020032785 - ANTONIO BITTENCOURT PAGOTTO

COM242 - SISTEMAS DISTRIBUÍDOS

Prof. Rafael Frinhani

INSTITUTO DE MATEMÁTICA E COMPUTAÇÃO
UNIFEI - Itajubá

2023

UNIVERSIDADE FEDERAL DE ITAJUBÁ
INSTITUTO DE MATEMÁTICA E COMPUTAÇÃO
SISTEMAS DE INFORMAÇÃO
COM242 - SISTEMAS DISTRIBUÍDOS

Uma Solução Descentralizada para o Compartilhamento Seguro de Mídias em uma Agência de Marketing Colaborativa

1 Introdução

A comunicação distribuída de arquivos funciona como solução para problemas aonde é necessário ter acesso de várias localidades diferentes. Neste trabalho será apresentado uma solução utilizando sistemas distribuídos peer-to-peer para compartilhamento de arquivos entre máquinas em redes LAN diferentes.

1.1 Contextualização

O problema apresentado na disciplina consiste em um sistema para compartilhamento de mídias digitais dentro de uma agência de marketing. A agência em questão possui mais de 70 filiais em todo o Brasil e trabalha com regime colaborativo onde materiais são usados por mais de uma filial. O grande motivo para desenvolvimento deste sistema é ser uma plataforma interna, o que gera um sigilo maior sobre as campanhas de marketing.

1.2 Objetivo geral

O objetivo geral deste trabalho é apresentar os métodos de implementação da solução do problema, fazendo com que seja possível entender de forma simples, cada passo que será apresentado no trabalho e possibilitando até a replicação da implementação.

1.3 Metodologias

Para este trabalho foram usadas tecnologias atuais e que fossem de fácil implementação, para que conseguíssemos um resultado mais rápido. Foram usadas tecnologias como Docker, Python com a biblioteca fast API, Banco de Dados orientado a arquivos. Todos estes pontos serão apresentados no decorrer do projeto.

1.4 Estrutura de capítulos

Este trabalho foi dividido em um total de 5 capítulos, sendo este o introdutório, na sequência terão capítulos falando sobre as tecnologias utilizadas no sistema, arquitetura implementada, descrição dos resultados e sistema final e uma conclusão sobre o conjunto deste trabalho.

2 Tecnologias Envolvidas no Projeto

Para se desenvolver um projeto escalável, flexível e rápido é necessário escolher as ferramentas corretas que auxilie a equipe de desenvolvimento no decorrer do projeto.

Com base na experiência da equipe e das necessidades do problema foram selecionadas as seguintes tecnologias, as quais irão ser abordadas uma a uma a seguir:

2.1 Docker

O Docker é uma plataforma de virtualização de nível de sistema operacional que permite empacotar, distribuir e executar aplicativos em contêineres. Esses contêineres fornecem uma maneira consistente de empacotar software e todas as suas dependências, como bibliotecas e configurações, em uma única unidade portável. O Docker será usado em cada uma das máquinas a fim de criar um ambiente igual entre as máquinas para facilitar a usabilidade e escalabilidade do projeto.

2.2 Phyton com FastAPI

Para o desenvolvimento da solução, será usado a linguagem de programação Phyton com o framework Fast API, que trará alguns benefícios no desenvolvimento do projeto, como os listados abaixo:

Velocidade de desenvolvimento: FastAPI é conhecido por sua produtividade e velocidade de desenvolvimento. Ele oferece uma sintaxe concisa e fácil de usar, que permite criar APIs de forma rápida e eficiente. O suporte a anotações de tipo do Python também simplifica a validação de dados e o processo de documentação da API.

Desenvolvimento assíncrono: O FastAPI é construído com base no Starlette, um framework assíncrono. Isso significa que você pode escrever código assíncrono para construir APIs altamente eficientes e escaláveis. O suporte a corrotinas assíncronas do Python, utilizando a sintaxe async/await, permite lidar com solicitações simultâneas e melhorar a capacidade de resposta da API.

Desempenho: O FastAPI é conhecido por sua excelente performance. Ele é construído sobre o poderoso framework assíncrono de alto desempenho chamado uvicorn, que utiliza o servidor de aplicativos ASGI. Essa combinação permite que o FastAPI lide com um grande número de solicitações simultâneas com eficiência.

Tipagem estática: O Python suporta anotações de tipo, e o FastAPI aproveita isso para validar os dados recebidos e retornados pela API em tempo de execução. Isso melhora a segurança e a robustez da API, permitindo que você detecte erros de tipo antes mesmo de executar o código.

Documentação automática: O FastAPI possui um sistema de geração automática de documentação interativa, o

que significa que a documentação da API é criada automaticamente com base nas anotações de tipo e nas informações fornecidas, gerando um arquivo .yaml, o qual será usado pelo Swagger para executarmos o projeto. Nesse projeto, a documentação automática será usada como interface gráfica de interação com a aplicação.

Ecossistema Python: Ao usar o FastAPI, você se beneficia do vasto ecossistema de bibliotecas e ferramentas disponíveis para Python. Há uma ampla variedade de bibliotecas para autenticação, autorização, manipulação de bancos de dados, testes, entre outros, que podem ser facilmente integradas às suas APIs FastAPI.

2.3 Banco de dados PostgreSQL

Para armazenar os dados do projeto, será utilizado o PostgreSQL, um sistema de gerenciamento de banco de dados relacional. O PostgreSQL oferece várias vantagens e recursos adequados para o projeto P2P.

Modelo relacional: O PostgreSQL é um banco de dados relacional, o que significa que ele segue o modelo de armazenamento de dados em tabelas com relações definidas entre elas. Isso é especialmente útil quando há necessidade de estruturar e organizar dados complexos e estabelecer relações entre entidades no projeto.

Transações ACID: O PostgreSQL suporta transações ACID (Atomicidade, Consistência, Isolamento e Durabilidade), garantindo que as operações no banco de dados sejam consistentes e duráveis. Isso é importante em um ambiente P2P, onde vários nós podem atualizar e acessar dados simultaneamente. As transações ACID ajudam a manter a integridade dos dados e garantem que as operações sejam realizadas de forma confiável.

Consultas SQL: O PostgreSQL é conhecido por sua poderosa linguagem de consulta SQL (Structured Query Language). Através do SQL, é possível realizar consultas complexas, realizar junções de tabelas, filtrar dados e executar operações de agregação. A familiaridade com a linguagem SQL facilita o desenvolvimento e a manipulação dos dados no projeto.

Confiabilidade e estabilidade: O PostgreSQL é conhecido por sua confiabilidade e estabilidade. Ele possui um histórico comprovado de robustez e é amplamente utilizado em ambientes de produção. Isso é essencial em um projeto P2P, onde é necessário garantir que o banco de dados seja confiável e possa lidar com cargas de trabalho variáveis e intensas.

Suporte a extensões: O PostgreSQL suporta extensões, o que permite estender a funcionalidade padrão do banco de dados com recursos adicionais. Existem várias extensões disponíveis que podem ser úteis em projetos P2P, como extensões para trabalhar com dados geoespaciais, busca de texto completo, armazenamento de dados JSON/BSON e muito mais.

2.3.1 Esquema do banco

Cada Nó contará com um banco de dados Postgres próprio, rodando em um contêiner docker dedicado, seguindo a estrutura da figura 1.

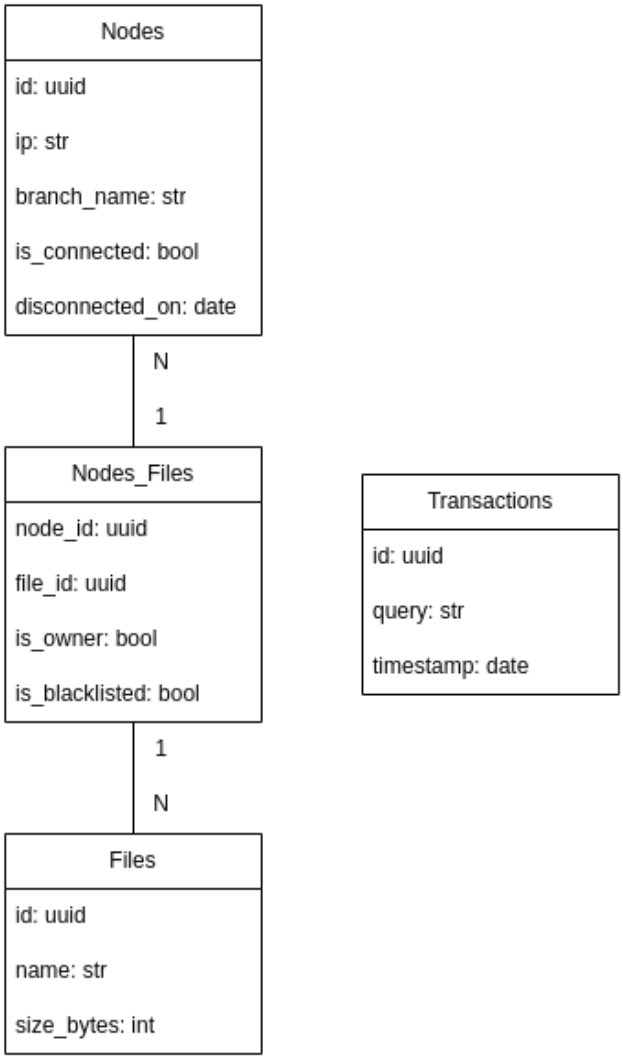

Figura 1: Esquema do banco de um Nó.

3 Arquitetura

3.1 Peer-to-peer (P2P)

É um tipo de arquitetura de rede na qual os dispositivos conectados têm papéis equivalentes e compartilham recursos diretamente entre si, sem a necessidade de um servidor centralizado. Nessa rede, cada dispositivo atua tanto como cliente quanto como servidor, permitindo que os participantes compartilhem e troquem informações diretamente uns com os outros.

Em uma rede P2P, não existe uma hierarquia rígida de dispositivos, ao contrário das redes cliente-servidor tradicionais, em que os servidores desempenham um papel central na distribuição de dados para os clientes. Em vez disso, cada dispositivo conectado na rede P2P pode atuar como um nó que contribui com recursos, como banda larga, capacidade de armazenamento ou poder de processamento, além de acessar os recursos compartilhados pelos outros nós.

Os dados e serviços são distribuídos entre os nós participantes da rede P2P, o que permite maior escalabilidade, resiliência e redundância. Essa arquitetura é frequentemente utilizada em aplicativos de compartilhamento de arquivos, como BitTorrent, em que os usuários compartilham partes de um arquivo diretamente entre si, sem depender de um servidor cen-

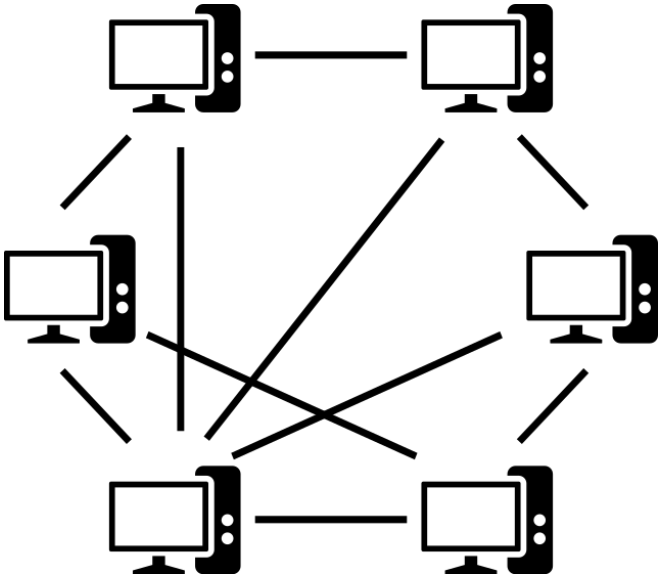

Figura 2: Exemplo visual de uma rede peer-to-peer.

tral. Outros exemplos incluem sistemas de compartilhamento de conteúdo, como redes sociais descentralizadas ou plataformas de streaming de vídeo P2P.

Essa estrutura é adequada para o cenário da aplicação visto que o objetivo que corta custos com serviços de armazenamento em nuvem previne problemas de segurança de dados sigilosos, visto que somente os nós presentes na rede, representados pelas filias da empresa, têm acesso aos aquivos, diferente de uma rede centralizada, onde esses arquivos seriam armazenados em um servidor de um provedor terceiro.

3.2 TCP e UDP

TCP (Transmission Control Protocol) e UDP (User Datagram Protocol) (TCP) são dois dos principais protocolos de transporte utilizados em redes de computadores para transmitir dados entre dispositivos. Eles diferem em termos de funcionalidade, confiabilidade e desempenho.

TCP é um protocolo orientado a conexão e confiável. Ele garante a entrega ordenada e livre de erros dos dados, além de lidar com o controle de congestionamento e a retransmissão de pacotes perdidos. O TCP estabelece uma conexão antes de iniciar a transmissão de dados, verificando se o destino está disponível e pronto para receber. Ele segmenta os dados em pacotes e os reordena no destino, se necessário, para recriar a sequência original.

UDP, por outro lado, é um protocolo orientado a datagrama e não confiável. Ele é mais simples e leve do que o TCP, não estabelecendo uma conexão prévia antes da transmissão de dados. O UDP envia pacotes de dados, chamados datagramas, de forma independente, sem garantias de entrega ou ordem. Ele não realiza retransmissões automáticas, controle de congestionamento ou correção de erros. Essa simplicidade faz com que o UDP seja mais rápido e eficiente em comparação com o TCP.

3.2.1 Comparação entre TCP e UDP

- **Confiabilidade:** O TCP é confiável, garantindo a entrega ordenada e sem erros dos dados, enquanto o UDP não oferece garantias de entrega ou ordem.

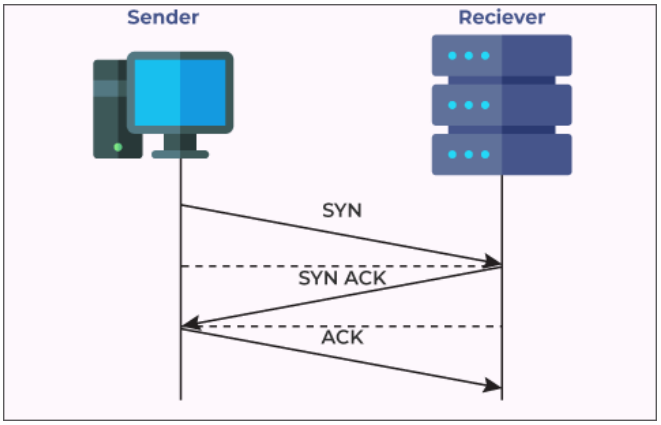

Figura 3: Exemplo de transmissão TCP.

- **Overhead:** O TCP possui um maior overhead devido aos mecanismos de controle e confiabilidade implementados, o que pode resultar em maior latência e consumo de recursos de rede. O UDP é mais leve, com um menor overhead, o que o torna mais rápido e eficiente.
- **Aplicações:** O TCP é amplamente utilizado em aplicativos que exigem transferência de dados confiável e ordenada, como transferências de arquivos e navegação na web. O UDP é comumente usado em aplicativos que priorizam a velocidade e o tempo real, como streaming de mídia, videoconferências e jogos online.
- **Exemplo de uso:** Um exemplo de uso do TCP é ao baixar um arquivo da Internet, onde a integridade dos dados é fundamental. Já o UDP é usado em streaming de vídeos ao vivo, onde pequenas perdas de pacotes não são críticas, mas a latência deve ser minimizada.

  Em resumo, o TCP é confiável, mas mais lento, enquanto o UDP é rápido, mas não confiável. A escolha entre TCP e UDP depende das necessidades específicas do aplicativo em termos de confiabilidade, velocidade e tolerância a erros.

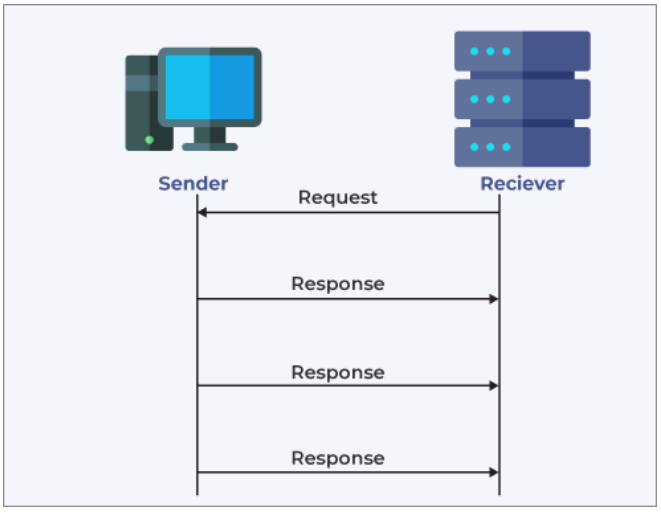

Figura 4: Exemplo de transmissão UDP.

3.3 Utilizar TCP ou UDP?

Aplicações de compartilhamento de torrents, como o BitTorrent, geralmente utilizam ambos os protocolos UDP e TCP em sua operação. A escolha entre UDP e TCP depende das diferentes fases e requisitos da transferência de dados.

No BitTorrent, o protocolo TCP é comumente usado para tarefas que exigem confiabilidade e integridade dos dados, como a comunicação inicial para estabelecer a conexão entre os pares (peers) e para baixar os metadados do arquivo (arquivo .torrent). O TCP é adequado para essas tarefas, pois garante a entrega ordenada e livre de erros dos pacotes de dados.

No entanto, após o estabelecimento da conexão e a obtenção dos metadados, o BitTorrent faz uso do protocolo UDP para a transferência eficiente dos dados reais do arquivo. O UDP é preferível nessa etapa porque oferece menor overhead e menor latência em comparação com o TCP. A transferência de dados em um ambiente P2P, geralmente envolve uma grande quantidade de pacotes e a velocidade é priorizada em relação à confiabilidade absoluta. Portanto, o UDP é utilizado para obter um desempenho mais rápido e uma comunicação mais eficiente durante a troca direta de partes do arquivo entre os pares.

Em nossa aplicação seguiremos as diretrizes do BitTorrent, utilzando o TCP para a conexão inicial entre nós, por exemplo, entre outros processos que necessitem de uma confiabilidade e integridade de dados e/ou troca de informações. Já o UDP será utilizado na transferência dos dados.

3.4 Hole Punching

É uma técnica utilizada para estabelecer comunicações diretas entre dois dispositivos em redes que possuem restrições de NAT (Network Address Translation) (hol). O NAT é uma tecnologia comum em roteadores e firewalls que permite compartilhar um único endereço IP público entre vários dispositivos em uma rede local.

Em uma rede descentralizada de compartilhamento de arquivos, em que os dispositivos estão conectados diretamente uns aos outros sem a necessidade de um servidor central, o hole punching é frequentemente utilizado para superar as limitações impostas pelos dispositivos NAT e permitir a comunicação direta entre os pares, caso contrário, cada par precisaria de configurações adicionais para configurar um ip público afim de conseguir se comunicar com outros pares.

O processo ocorre da seguinte forma:

1. Cada dispositivo (ou nó) na rede P2P tenta iniciar uma conexão com outro dispositivo. No entanto, devido ao NAT, o roteador bloqueia as conexões iniciadas externamente.

2. Cada dispositivo envia pacotes para o endereço IP e porta do outro dispositivo, mas os pacotes são bloqueados pelo NAT.

3. Para contornar o bloqueio do NAT, os dispositivos utilizam um servidor intermediário (servidor de relé) fora da rede local para ajudar na troca de informações. Os dispositivos se comunicam com o servidor intermediário usando TCP, que geralmente não é bloqueado pelo NAT.

4. Os dispositivos compartilham informações sobre o endereço IP e a porta atribuída pelo NAT ao servidor intermediário.

5. Os dispositivos repetem o envio de pacotes um para o outro, agora incluindo as informações do servidor intermediário. Isso faz com que o NAT registre as informações do servidor intermediário e abra uma "porta"temporária que permite a comunicação direta entre os dispositivos.

6. Após a "perfuração de buraco"ser bem-sucedida, os dispositivos podem estabelecer uma conexão direta, ignorando o servidor intermediário.

Dessa forma, o hole punching permite que os dispositivos P2P superem as restrições do NAT e estabeleçam comunicações diretas, essenciais para uma rede de compartilhamento de arquivos descentralizada. Isso possibilita a transferência de dados entre os dispositivos sem a necessidade de passar por servidores centrais, melhorando a eficiência e a velocidade do compartilhamento de arquivos entre os pares na rede.

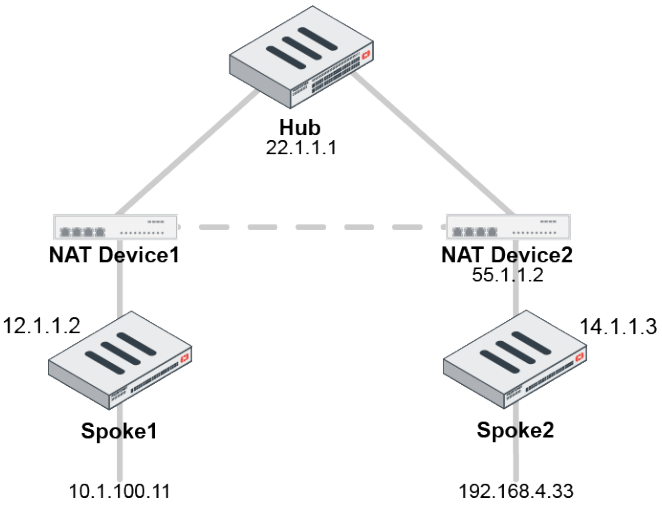

Figura 5: Esquema de Hole Punching, exemplo para UDP.

3.5 Blacklist

Uma blacklist em uma aplicação é uma lista de elementos indesejados ou proibidos que são bloqueados ou restringidos pelo sistema. Essa lista geralmente contém informações como endereços IP, nomes de domínio, URLs, palavras-chave ou qualquer outro tipo de identificador que represente conteúdo ou entidades a serem excluídas ou evitadas.

A função principal de uma blacklist é impedir que elementos indesejados acessem ou interajam com a aplicação. Isso pode incluir bloquear o acesso de usuários específicos, filtrar conteúdo impróprio, evitar a execução de determinadas ações ou restringir o uso de recursos específicos.

O esquema do banco de dados já está estruturando de forma a incluir essa funcionalidade para cara arquivo individualmente, por meio de um atributo na tabela Nodes_Files, que lista individualmente a relação de cada Nó com cada arquivo, para dizer se uma dessas relações está blacklisted ou não, ao mesmo tempo em que permite que um Nó que não pode ter acesso a um arquivo, caso os arquivos sejam criptografados, ainda possa armazená-lo na intenção de auxiliar a disponibilidade do arquivo na rede como um todo.

4 Solução

Nessa sessão será detalhado como funciona a conexão da rede e a gerência de arquivos por parte de cada um dos Nós.

A aplicação foi planejada para permitir o máximo de disponibilidade de todos os arquivos na rede. Levando em conta o pior caso, onde os nós donos do arquivo podem estar desconectados, têm-se o sistema de cópias do arquivo em outros Nós, mas somente o Nó dono do arquivo pode executar as operações de exclusão, atualização e compartilhamento seletivo do arquivo da rede.

4.1 Interface do Sistema

Para controlar o sistema, será criada uma interface de API usando o Swagger, que disponibilizará endpoints de uma API, a qual será responsável pela execução do sistema.

1. POST /register : Esse endpoint será usado quando um novo nó quiser entrar na rede, ele precisará passar o IP de um nó já membro da rede como parâmetro no body para que a interface identifique a rede a qual o usuário deseja entrar. O body desse endpoint deverá conter o parâmetro "ip", que será o IP de algum nó já conectado a alguma rede existente.

2. POST /unregister : Esse endpoint será usado quando um novo nó quiser sair permanentemente da rede na rede. Esse endpoint não terá um body.

3. POST /connect : Esse endpoint será usado quando um nó, já membro da rede, quiser se conectar a rede. Quando esse endpoint é invocado, o nó o qual está se conectando envia uma requisição a todos os outros nós membros da rede, para notificar que esse nó agora está conectado, então o parâmetro "connected"do registro do nó o qual está se conectando é é alterado para "true"em todos os outros nós da rede. Esse endpoint não terá um body.

4. POST /disconnect : Esse endpoint será usado quando um nó, já membro da rede, quiser se desconectar a rede. Quando esse endpoint é invocado, o nó o qual está se desconectando envia uma requisição a todos os outros nós membros da rede, para notificar que esse nó agora está desconectado, então o parâmetro "connected"do registro do nó o qual está se conectando é é alterado para "false"em todos os outros nós da rede. Esse endpoint não terá um body.

5. POST /files : Nesse endpoint o usuário poderá adicionar novos arquivos a rede. Haverá um attachment do arquivo nesse endpoint, na interface do Swagger, a qual o usuário poderá selecionar o arquivo a ser inserido na rede. Após isso, a API irá notificar todos os outros nós que esse nó agora possui um novo arquivo, e o arquivo será salvo no diretório determinado pelo sistema localmente. Esse endpoint não terá um body.

6. GET /files: Esse endpoint é usado para o usuário visualizar os arquivos que estão disponíveis na rede. Quando essa interface é acionada, a API retorna todos os arquivos de cada nó e os coloca em uma lista única e retorna essa informação para o usuário. O response desse endpoint irá verificar qual Nó fez a requisição, e se esse nó estiver na blacklist com algum arquivo, esse arquivo então não será retornado nessa lista. Esse endpoint não terá um body.

7. PUT /files/fileId: Essa interface é responsável por atualizar um arquivo já existente na rede. O usuário necessitará do ID do arquivo para atualiza-lo, o qual o usuário terá acesso através do endpoint GET /files. Tendo esse ID, o usuário o inserirá no PUT como parâmetro e também irá fazer o upload do novo arquivo através da interface.

8. DELETE /files/fileId: Quando se quiser apagar um arquivo da rede basta acionar esse endpoint, neste caso basta passar o ID do arquivo e então o mesmo será apagado da rede. Esse endpoint não terá um body.

9. POST /files/blacklist: Esse endpoint será responsável por adicionar arquivos na blacklist. O body desse endpoint deverá conter os seguintes parâmetros: id do arquivo e usuário. Ou seja, o usuário poderá adicionar arquivos na blacklist um a um, enviando o usuário e arquivo os quais estarão restritos de tal visualização.

10. GET /files/blacklist: Através desse endpoint será possível ler todos os arquivos e usuários que estarão na blacklist. Caso o usuário que tenha feito a requisição esteja na blacklist ele não vai ter no retorno da requisição o registro referente ao seu usuário.

11. PUT /files/blacklist/fileId: Esse endpoint será responsável por modificar arquivos na blacklist. O body desse endpoint deverá conter o seguinte parâmetro: "usuário".

12. DELETE /files/blacklist/fileId: Neste caso esse endpoint irá remover algum arquivo da blacklist. Esse endpoint não terá body.

4.2 Entrada de um novo nó na rede

Para se iniciar a conexão, o Nó que deseja se conectar a rede necessita do IP de algum dos nós, e a partir disso ele envia uma requisição para esse IP através da API a qual cada um dos Nós terá rodando localmente.

O Nó requerente enviará uma requisição no endpoint POST /register , passando o IP de um Nó já conectado a rede como parâmetro. Após isso, será feita uma validação se o software do Nó requerente é válido ou não, a partir de uma autenticação JWT implícita dentro de cada uma das aplicações que estarão rodando nos Nós.

Caso a autenticação esteja errada, então a conexão é finalizada. Caso a autenticação esteja correta, então o Nó requerente recebe do Nó conectado a lista de todos os nós conectados em sua rede. Essas informações são salvas no banco de dados local do Nó requerente.

Após isso, o Nó requerente notifica cada um dos nós de sua base de dados que ele agora está conectado a essa rede, então todos os outros nós irão armazenar os dados do Nó requerente em seus respectivos bancos de dados.

4.3 Nó conhecido ficando online/offline

Na base de dados, ilustrada na sessão 2, pode-se reparar que no registro de cada um dos Nós temos o atributo "connected", que diz se o nó está conectado a rede ou não naquele instante.

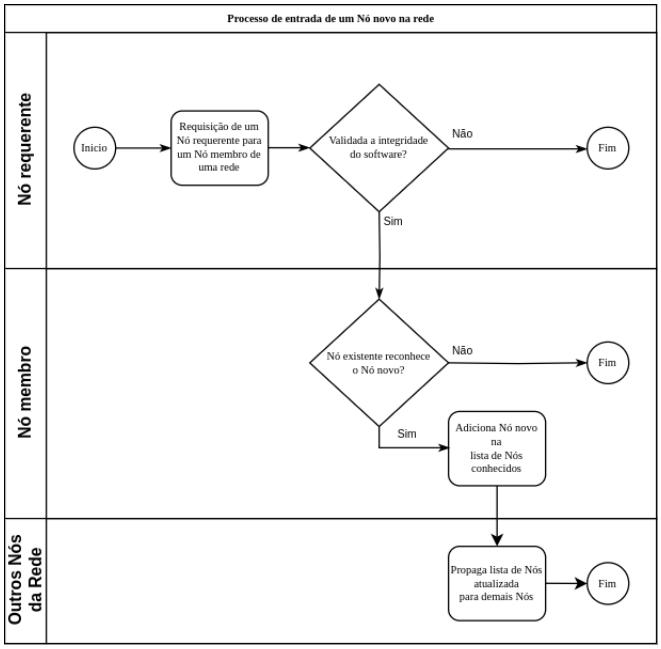

Figura 6: Processo de entrada de um novo nó na rede.

Quando um nó já conhecido se conecta a rede, através do endpoint POST /connect, ele notifica todos os nós contidos em sua base de dados que agora ele está conectado, então todos os outros nós mudarão o atributo "connected"desse respectivo nó para "true".

Já em caso de desconexão, o Nó o qual irá se desconectar necessita notificar todos os outros nós que ele está se desligando da rede. Isso é feito pelo endpoint POST /disconnect, quando esse endpoint é invocado, todos os Nós contidos na base de dados do nó que deseja se desconectar são notificados que esse nó agora está offline, mudando o parâmetro "connected"desse nó para "false".

Caso algum nó tente enviar alguma requisição para outro nó, que esteja com o parâmetro "connected"como "true", e essa requisição retorne um 404 Not Found, então o parâmetro "connected"desse nó será alterado para "false", siginifica que um nó tentou fazer conexão com esse respectivo nó e não conseguiu, ou seja, esse nó pode ter se desconectado sem notificar os outros, por algum problema de conexão.

4.4 Inserção de um novo arquivo na rede

Para se inserir um novo arquivo na rede tem-se o endpoint POST /files. Quando esse endpoint é invocado, será feita uma consulta no banco de dados local do Nó que invocou esse endpoint, então é buscado cada IP dos nós conectados a rede, e é enviada uma requisição para cada nó notificando que o nó o qual fez o upload de arquivos agora tem esse arquivo salvo em sua base de dados, atualizando a base de dados de todos os outros membros da rede.

4.5 Atualização de arquivos

A atualização de um arquivo é feita também através do endpoint PUT /files/fileId. O procedimento é similar ao de inserção de arquivo, a diferença é que deve-se passar o ID do arquivo como path param, e o arquivo é anexado no request da interface, no Swagger.

Se o arquivo a ser atualizado existe apenas no nó o qual está fazendo o upload, então o arquivo é atualizado localmente. Mas se o arquivo existir em outro nó, então é feito o envio desse arquivo para o nó o qual já contém esse arquivo para que seja feita a atualização do arquivo existente.

4.6 Leitura de arquivos

Assim como já mencionado na sessão 3.1 deste documento, dentro da arquitetura Peer-to-peer, todos dispositivos estão conectados entre si, devido a isso, é feito o compartilhamento do arquivo por partes.

Dentro da rede do sistema, há a possibilidade de mais de um dispositivo possuir um arquivo. Quando um dos dispositivos em questão deseja este arquivo, deve verificar em seu banco quais nós possuem o arquivo e solicitar para os mesmos. Após isso, os nós solicitados irão iniciar o envio do arquivo de forma particionada e todos ao mesmo tempo. Este método irá facilitar o envio, deixando mais rápido e fazendo com que o dispositivo destinatário consiga montar o arquivo final de acordo com que as partes vão chegando.

A leitura de arquivos é feita através do endpoint GET /files, mencionado na sessão 4.1, onde o usuário fará uma requisição nesse endpoint, e então o fluxo seguirá como no passo a passo abaixo:

1. Nó verifica em seu banco quais outros nós possuem o arquivo;
2. Nó solicita o arquivo para os N nós que possuem o arquivo;
3. Os N nós recebem a solicitação;
4. Os N nós encontram o arquivo e particionam para envio;
5. Os N nós enviam simultaneamente o arquivo;
6. O nó receptor monta o arquivo no que os pacotes vão chegando na ordem.

4.7 Exclusão de arquivo

Quando há a exclusão de um arquivo na rede, como há mais de uma cópia em diferentes nós, é necessário ocorrer a deleção em todos nós. Para isto, é necessário que o nó que iniciou a exclusão verifique quais outros nós possuem este arquivo e envie um aviso para exclusão. A exclusão é feita a partir do endpoint DELETE /files, mencionado na sessão 4.1. Para maior entendimento, será feito um passo a passo com o processo de exclusão:

1. Nó apaga o arquivo localmente;
2. Verifica no banco os outros nós que possuem o arquivo;
3. Manda notificação para deleção do arquivo;
4. Atualiza lista de nós;
5. Propaga a lista de nós atualizada.

4.8 Compartilhamento seletivo

Dentro de uma empresa, muitas vezes é necessário certo nível de sigilo, para isso, há a necessidade de implementação de uma blacklist.

Para adicionar um nó na blacklist usa-se o endpoint POST /files/blacklist , e então deve-se passar como parâmetro no body o id do arquivo e o id do usuário o qual estará restrito de visualizar este arquivo.

Pode-se também remover algum usuário da blacklist de um tal arquivo através do endpoint DELETE /files/blacklist/-fileId.

Conforme já contemplado na estrutura do banco de dados, a tabela de relacionamento entre arquivos e Nós também implementa o atributo booleano de is_blacklisted para cada registro individual desse relacionamento.

4.9 Disponibilidade dos arquivos

A disponibilidade dos arquivos em uma rede peer-to-peer descentralizada de compartilhamento de dados é um fator crucial para o sucesso do sistema. Ao analisar a melhor abordagem para garantir essa disponibilidade, consideramos a distribuição das cópias dos arquivos entre os nós da rede.

Acreditamos que ter aproximadamente um terço dos nós da rede com uma cópia do arquivo é uma estratégia eficaz para manter a disponibilidade dos arquivos quase constante. Nossa decisão se baseia no entendimento de que a indisponibilidade de um terço dos nós é menos provável de ocorrer simultaneamente, o que aumenta a chance de ter pelo menos uma cópia disponível.

Ao adotar essa estratégia de distribuição das cópias, podemos mitigar o impacto da indisponibilidade de nós individuais na rede como um todo. Em vez de depender exclusivamente de um único nó para acessar um determinado arquivo, aproveitamos a redundância das cópias distribuídas para manter a disponibilidade, mesmo em situações adversas.

4.10 Sincronização dos bancos

No projeto, utilizamos um sistema de armazenamento de transações baseado em uma tabela no banco de dados PostgreSQL. Cada nó da rede mantém uma instância idêntica do banco de dados. As transações são armazenadas na tabela em ordem cronológica, com base no carimbo de data/hora (timestamp) em que ocorreram.

O objetivo desse sistema é garantir a consistência dos dados entre os nós, mesmo considerando o atraso de propagação das operações pela rede. Embora os atributos do banco de dados possam não estar sempre definidos, como nomes de arquivos ou filiais da empresa, nenhuma operação envolve cálculos matemáticos. Portanto, desde que as transações sejam executadas na ordem cronológica correta, o resultado final será sempre correto.

Para lidar com a situação em que um nó A realiza uma operação em uma entidade e propaga essa alteração para os bancos de dados dos outros nós, enquanto o nó B realiza uma operação na mesma entidade após a operação do nó A, mas antes de receber a operação do nó A devido ao atraso na propagação da rede, adotamos a seguinte abordagem:

1. Registro de transações: Cada nó registra suas transações na tabela do banco de dados, indicando o carimbo de data/hora (timestamp) correspondente.

2. Propagação de transações: Os nós propagam suas transações para os outros nós da rede, permitindo que todas as instâncias do banco de dados sejam atualizadas de forma consistente.

3. Verificação de ordem cronológica: Quando um nó recebe uma nova transação, ele verifica o carimbo de data/hora (timestamp) da transação recebida e o compara com a última transação registrada em seu banco de dados. Se o timestamp da nova transação for anterior ao da última transação registrada, a nova transação é executada no banco de dados local.

4. Execução de transações: Todas as transações com timestamps subsequentes à última transação registrada no banco de dados são executadas em ordem cronológica. Dessa forma, garante-se que as operações sejam aplicadas corretamente e que o estado final do banco de dados seja consistente em todos os nós da rede.

5. É importante ressaltar que essa abordagem depende da sincronização adequada entre os nós para garantir que as transações sejam propagadas e recebidas corretamente. Além disso, a corretude do resultado final depende da consistência dos timestamps registrados em cada nó e da correta ordenação das transações. Essa estratégia visa minimizar conflitos e inconsistências, garantindo a integridade dos dados em um ambiente de rede distribuída.

4.11 Deleção de nó

Quando um nó deseja sair da rede e não apenas desligar, para religar mais tarde, é necessário certo cuidado para que não haja deleções más intencionadas, realizadas por atacantes ou por erro de uso da aplicação. O melhor caso levantado para esta situação foi a de a deleção apenas ocorrer por solicitação do próprio nó. Após a notificação de todos nós, é feita a transferência de propriedade de arquivo, que será debatida na próxima sessão, e por fim, o nó pode ser excluído da rede, sem que haja a possibilidade de novas solicitações para o mesmo.

4.12 Repasse de propriedade do arquivo

Antes da deleção de um nó, é necessário que haja a mudança da propriedade do arquivo. Levando em consideração que há mais de uma cópia na rede, pela questão de disponibilidade e redundância de arquivos, o processo de repasse de propriedade funcionará da seguinte forma:

1. O nó que irá ser deletado manda notificação para todos outros nós avisando de que será deletado;

2. Os outros nós verificarão quais deles possuem os arquivos presentes no nó a ser deletado;

3. É feito um sorteio entre os nós possuidores do arquivo, para ver qual será eleito o novo dono do arquivo.

4. A atualização de dono é propagada pela rede, para que todos nós atualizem seus bancos de dados.

5 Conclusão

Em conclusão, o processo de planejamento do nosso projeto foi desafiador, mas extremamente enriquecedor. Identificamos as principais dificuldades, como a sincronização dos nós, a integridade dos dados e a definição de casos de uso relevantes para a empresa. No entanto, com dedicação, colaboração e pesquisa, conseguimos encontrar soluções eficazes para esses desafios.

Uma das facilidades encontradas foi a escolha das tecnologias. Optamos por utilizar Python como linguagem de desenvolvimento devido às suas vantagens, como a vasta comunidade de desenvolvedores e as bibliotecas disponíveis. O PostgreSQL foi escolhido como banco de dados por ser uma tecnologia familiar e adequada para a manipulação de dados. O uso do Docker também nos proporcionou facilidades, permitindo o isolamento do ambiente de desenvolvimento.

Ao superar as dificuldades e aproveitar as facilidades, alcançamos um planejamento sólido para a nossa rede peer-to-peer. Estamos confiantes de que as medidas tomadas para garantir a sincronização dos nós, a integridade dos dados e a implementação de funcionalidades relevantes atenderão às necessidades da empresa. Acreditamos que estamos prontos para iniciar a próxima fase do projeto, a implementação, com um plano bem definido e estratégias eficientes para enfrentar os desafios que ainda possam surgir.

6 Divisão do trabalho

- **ANTONIO:** 2, 2.1, 2.2, 4, 4.1, 4.2, 4.3, 4.4, 4.5
- **HENRIQUE:** 2.3, 3, 3.1, 3.2, 3.3, 3.4, 4.8, 4.9, 5
- **MARCELO:** 1, 1.1, 1.2, 1.3, 1.4, 4.6, 4.7, 4.8, 4.11, 4.12

Referências

[P2P] Implementing peer-to-peer data exchange in python. Disponível em: https://medium.com/@luishrsoares/implementing-peer-to-peer-data-exchange-in-python-8e69513489af.

[TCP] Tcp vs udp: Differences between the protocols. Disponível em: https://www.avast.com/c-tcp-vs-udp-difference#:~:text=The%20main%20difference%20between%20TCP,reliable%20but%20works%20more%20quickly.

[hol] Udp hole punching. Disponível em: https://en.wikipedia.org/wiki/UDP_hole_punching.