UFABC

Sistemas Distribuídos - Projeto de Programação

Napster

Nome: Wesley Lima de Araujo
RA: 11201721514

2º quadrimestre de 2022
Santo André, SP

Vídeo com demonstração de uso: https://youtu.be/mVoHes2NXso

1. Objetivo do Projeto e Definição do Sistema

O objetivo do projeto é criar um sistema P2P que permita a transferência de arquivos gigantes (mais de 1 GB) entre peers, intermediados por um servidor centralizado, utilizando TCP e UDP como protocolos da camada de transporte. O sistema deve funcionar de maneira similar, mas muito reduzida, ao sistema Napster, um dos primeiros sistemas P2P. Nas próximas seções desse trabalho faremos as apresentações das funcionalidades do sistema implementado pelo aluno. Primeiro vamos apresentar as funcionalidades do servidor e depois dos peers, explicando em "alto nível" seu funcionamento com base no código desenvolvido. Depois explicaremos o esquema de threads do sistema usando um diagrama. Sobre a transferência de arquivos gigantes também dedicaremos uma seção a explicar como foi feito o processo. Lembrando que a existência de apenas três arquivos *.java* está sendo considerada: Servidor.java, Peer.java e Mensagem.java.

2. Classe Mensagem

Antes de começarmos a explicar as funcionalidades do servidor e dos peers, gostaria de comentar sobre a organização e funcionamento da classe responsável pela padronização de comunicação entre essas diferentes entidades do nosso sistema, a classe Mensagem.

A classe Mensagem é uma das três classes previstas no desenvolvimento do projeto. Ela é usada tanto pelo servidor quanto pelos peers para criar as mensagens enviadas visando comunicação com outros objetos. Por exemplo, um peer que deseja fazer JOIN deve criar um objeto da classe mensagem com parâmetros (1, "JOIN"), transformá-lo em um formato JSON e enviar para o servidor. O servidor, quando receber essa mensagem, também vai compor uma mensagem para responder o peer, porém os parâmetros de construção são (1, "JOIN_OK").

Existem três métodos construtores para essa classe. O primeiro, visto na linha *33*, recebe como parâmetros um inteiro (o ID da mensagem) e uma string (o conteúdo da mensagem). O ID passado deve ser um dos padronizados na variável privada da classe ids, do tipo Map<int, String>, definida entre as linhas *14* e *26*, que identifica através do ID qual o tipo de comunicação que será feita. Esse primeiro método construtor é usado na grande maioria das composições de mensagem que serão enviadas. Só reforçando, existem dois atributos internos da classe que armazenam o ID e o conteúdo passado (id e conteudo).

O segundo método construtor, que pode ser visto entre as linhas *39* e *43*, têm estrutura muito semelhante a do primeiro construtor, porém, aqui o segundo parâmetro não é um String de conteúdo, mas sim uma ArrayList de Strings. Para esse construtor essa lista passada é o conteúdo da mensagem. Esse segundo método é especialmente usado na devolutiva da requisição SEARCH.

Por fim, temos um terceiro método construtor visto entre as linhas *69* e *87*, sua grande diferença é que enquanto os outros métodos são usados na parte do envio da mensagem este método é usado na transformação de uma String em formato JSON num objeto da classe. O método em si recebe como argumento um JSONObject (biblioteca JSON.org) e constrói a partir dos campos desse JSON nosso objeto.

Como já dito, a classe mensagem usa muito a seu favor o formato JSON para padronização. Pense que qualquer mensagem enviada ou recebida é um array de Bytes, esse array de Bytes é convertido numa String. Aqui, sempre é esperado que a String esteja formatada para representar um objeto JSON, dessa forma podemos usar um método construtor de objetos JSON (nativo da JSON.org) para criar um JSONObject com os campos desejados.

Por ter essa relação com o formato JSON, foram implementados na classe Mensagem dois métodos visando a conversão de um objeto instanciado da classe Mensagem em um JSONObject, assim podemos obter a representação da mensagem como JSON de forma fácil e que facilite a conversão na String que será enviada para a outra ponta da comunicação. Ambos os métodos não recebem argumentos na chamada e devolvem um JSONObject.

O primeiro desses métodos é chamado de criarJSON, esse primeiro método se encontra na linha *45*. O segundo é o método criarJSONLista na linha *55*. A grande diferença entre esses dois métodos citados é que é o criarJSONLista é usado para conversão quando o conteúdo da mensagem se encontra na variável conteudoLista. Ou seja, quando o conteúdo da mensagem é uma lista, como no caso de uma requisição SEARCH, onde o servidor devolve uma lista de peers para o solicitante. Já o criarJSON é usado quando o conteúdo da mensagem é uma String comum.

id	significado/objetivo da mensagem
1	JOIN
2	LEAVE
3	SEARCH
4	UPDATE
5	ALIVE
6	DOWNLOAD
7	DOWNLOAD_NEGADO

Tabela com significados dos IDs de mensagem.

3. Classe Servidor

No desenvolvimento desse projeto a classe servidor foi desenvolvida para ser um centralizador de informações sobre a rede de peers atual. Para realizar atualizações dessa rede, o Servidor atende a requisições peers relacionadas à entrada, saída e atualização da rede. Para realizar essa função o Servidor implementado usa um banco de dados SQLite para armazenar as informações. Diferente do Python que já possui SQLite nativo, para uso no Java foi necessário usar um *.jar* externo para implementar o Servidor com SQLite. O *.jar* usado está sendo enviado junto com o resto do projeto.

Dito isso, existem algumas classe aninhadas dentro da classe Servidor responsáveis por funções diversas, mas especialmente relacionadas ao atendimento de requisições através de threads e atualização do banco de dados em uma interface padrão. Segue uma breve descrição de cada classe, elas serão mais bem detalhadas mais pra frente nessa mesma seção:

- **Classe BancoServidor -** Classe responsável por implementar os métodos de interface com o Banco de Dados da aplicação. Então, quando um peer faz JOIN, LEAVE, UPDATE ou alguma outra requisição, o Servidor atualiza o Banco de Dados através de métodos dessa classe. Na prática, os métodos dessa classe consistem na aplicação de queries na tabela usada para manter as informações do sistema. Classe contida entre as linhas *79* e *266.* Trata-se de uma classe que usa o construtor padrão do java.
- **Classe ThreadAtendimento -** Classe que trata de fato as requisições dos peers. Quando um peer envia um request ao servidor na porta já conhecida, uma nova thread dessa classe é lançada para realizar os processamentos necessários. Todas as informações necessárias para realizar o tratamento da requisição estão contidas no próprio pacote datagrama enviado pelo peer. De maneira bem resumida, a classe é um grande if, que dependendo o id da mensagem recebida dos peers vai tomar algumas ações. Classe contida entre linhas *269* e *432.* Essa classe possui só um construtor, esse exige o DatagramPacket recebido e o conector com o Banco de Dados da aplicação. Naturalmente essa classe estende Thread.
- **Classe ThreadAlive -** Thread responsável por consultar os peers atualmente em estado de JOIN e enviar o request ALIVE para eles. Trata-se de uma thread exclusiva para fazer esse envio de mensagens e para atualizar a tabela de dados, removendo peers que não tenham enviado o ALIVE_OK em até 30 segundos após o disparo da ALIVE. Também possui só um método construtor, cujo único parâmetro é um conector para o Banco de Dados da aplicação. Essa é outra classe que estende Thread.

Dito o que as classes aninhadas fazem, também é importante ressaltar qual a função exercida pelo método Main da classe Servidor. O método principal implementado inicia sempre criando a conexão com o banco de dados e limpando os dados contidos nele anteriormente, entre linhas *32* e *37*. Após isso cria a threadAlive, responsável por ficar enviando as mensagens de ALIVE para todos os peers que estão na rede naquele momento, linhas *40* e *41*. Após essas primeiras etapas, o resto do método Main é bem simples, trata-se de um loop while infinito que fica escutando na porta 10098 os requests UDP dos peers. Para cada request recebido uma nova thread de atendimento é lançada para processar o pedido.

Explicada a estrutura geral da classe Servidor, vamos falar dos tratamentos de requisições realizados dentro da classe aninhada ThreadAtendimento. Só destacando que a primeira coisa feita quando o método start é chamado é a tradução dos dados do pacote em String, depois em JSON e depois em um objeto da classe mensagem. Os casos de atendimento das requisições são baseados no atributo idMensagem da classe Mensagem. Segue uma breve explicação do que é feito em cada caso:

- **Se idMensagem é 1 -** Nessa situação algum peer está fazendo a solicitação de JOIN. A primeira coisa que a ThreadAtendimento faz é parsear o conteúdo da mensagem recebida. O conteúdo da mensagem contém os arquivos que o peer quer disponibilizar para o sistema. Para cada arquivo é adicionada uma linha na tabela do Banco de Dados que diz que um certo IP, numa certa porta contém aquele arquivo. Depois disso é enviada uma mensagem de JOIN_OK para o mesmo endereço que enviou o pacote original. Importante ressaltar que o endereço registrado desse peer é retirado do DatagramPacket recebido. Tratamento feito entre as linhas *299* e *330*.
- **Se idMensagem é 2 -** Quando algum peer quer fazer o LEAVE. Nessa situação o conteúdo da mensagem é simplesmente a String “LEAVE” e o endereço desse peer. Assim, mesmo que o peer use outra porta para enviar a mensagem de LEAVE o servidor sabe qual dos peers da rede está saindo. Na prática, o que é feito é uma query de delete na tabela. Também é enviado para o peer uma mensagem de LEAVE_OK. Tratamento feito entre as linhas *333 e 359*.
- **Se idMensagem é 3 -** Solicitação SEARCH. Para o servidor chega uma mensagem com o nome do arquivo buscado como conteúdo. Após uma query de busca que retorna todos os peers ativos com esse arquivo, o servidor envia para o peer solicitante a lista dos IPs resultantes. Tratamento feito entre as linhas *361 e 391*.
- **Se idMensagem é 4 -** Solicitação de UPDATE, isso é, após um peer baixar um arquivo de outro peer, o primeiro peer envia essa mensagem de UPDATE com o nome do arquivo baixado para que o servidor atualize a tabela de informações. O Servidor faz a inserção de uma nova linha na tabela com os

dados de IP, porta e arquivo novo. A resposta enviada para o peer que enviou a mensagem é um UPDATE_OK. Implementado entre as linhas *394* e *420*.

- **Se idMensagem é 5 -** Resposta ALIVE_OK vinda de algum peer. Quando um peer ainda ativo recebe a mensagem de ALIVE enviada pela Thread Alive, ele devolve um ALIVE_OK. Quando o Servidor recebe essa mensagem ele atualiza a coluna de Alive desse peer na tabela, o que garante que o peer fique “vivo” até o próximo ciclo. Tratamento entre as linhas *423* e *429*.

Só lembrando que todas essas operações que afetam de alguma forma a tabela de dados usam algum método da classe BancoServidor. Falando nessa classe vamos entender melhor a modelagem dos dados envolvida e como ela facilita o trabalho da thread de ALIVE do servidor.

3.1 Uso do Banco de Dados

O Banco de Dados é de uso exclusivo do servidor para manter os registros de quais peers e arquivos disponíveis no sistema no momento. Só é utilizada uma tabela que possui o seguinte schema:

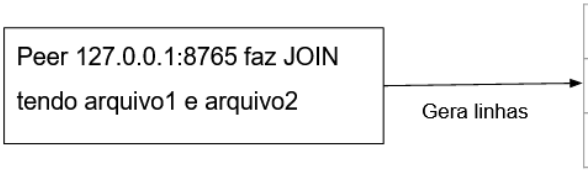
	Name	Data type
1	ip	TEXT
2	porta	TEXT
3	arquivo	TEXT
4	flag_alive	INTEGER

Colunas da tabela de dados usada pelo Servidor

Veja na imagem acima o nome e o tipo de dado das colunas usadas. O nome oficial da tabela no Banco de Dados é usuarios_arquivos. As colunas ip e porta referem-se ao ip e a porta do peer, enquanto a coluna arquivo indica que o aquele peer possui determinado arquivo.

Por fim, temos a coluna flag_alive, que como o nome indica funciona como um flag. A cada 30 segundos a Thread Alive escaneia a tabela e elimina as linhas onde o flag_alive possui valor 0. Depois dessa “limpeza” da tabela a Thread Alive atribui valor 0 na coluna flag_alive para todas as linhas da tabela e envia para cada peer ainda ativo a mensagem ALIVE. Aqueles que responderem a mensagem com ALIVE_OK fazem com que o valor de flag_alive nas linhas respectivas a eles volte a ser 1, assim, no próximo ciclo de “limpeza” da tabela garante-se que suas linhas não serão deletadas.

Segue um esquema de funcionamento de quando um peer faz JOIN:



ip	porta	arquivo	flag_alive
127.0.0.1	8765	arquivo1	1
127.0.0.1	8765	arquivo2	1

Agora que explicamos em alto nível o uso da tabela, vamos dar uma breve explicação sobre cada operação disponível para se realizar sobre essa tabela usando os métodos da classe BancoServidor. Como já dito, quase todo método dessa classe é a execução de uma query sobre a tabela usuarios_arquivos, com exceção apenas dos métodos de conexão e desconexão. Seguem as explicações:

- **criaTabela -** Não recebe parâmetros. Cria a tabela usuarios_arquivos Banco de Dados, caso ela já exista apaga os dados que lá existem e recria a tabela com mesmo schema mas sem os dados antigos. Entre as linhas 111 e 124.
- **peerJoin -** Recebe o ip do peer, a porta do peer e o nome de um arquivo. Insere uma nova linha na tabela com essas informações e com valor de **flag_alive** igual a 1. Esse método é chamado uma vez para cada arquivo passado durante o JOIN de um peer. Entre as linhas 127 e 141.
- **peerLeave -** Recebe o ip e a porta de um peer. Deleta todas as linhas com esses valores de ip e porta. Sempre chamado quando uma requisição de LEAVE é realizada. Implementado entre as linhas 144 e 153.
- **peerSearch -** Método que recebe o nome de um arquivo e devolve uma ArrayList com os endereços de peers que contém o arquivo passado. Usado na operação SEARCH. Implementado entre as linhas 156 e 173.
- **peerUpdate -** Quando um peer envia uma mensagem de UPDATE, esse método é chamado para adicionar à tabela uma linha específica para o arquivo novo com as informações desse peer. Recebe o nome do arquivo como parâmetro. Entre as linhas 176 e 189.
- **zerarAlive -** Método que não precisa de nenhum parâmetro. Zera toda a coluna de flag_alive para todas as linhas, assim, só os peers que enviarem mensagens ALIVE nos próximos 30 segundos, e consequentemente atribuir valor 1 a essa coluna, vão se manter na tabela. Implementado entre as linhas 192 e 200.
- **atualizarAlive -** Quando o Servidor recebe um ALIVE_OK de um peer esse método é invocado para setar valor 1 na coluna flag_alive, garantindo a permanência desse peer durante a próxima limpeza da tabela. Entre as linhas 202 e 211.
- **limparAlive** - Método responsável por fazer a limpeza dos peers que não atualizaram o flag_alive. Além disso, também devolve um ResultSet, que é o resultado de uma query. A query executada busca as linhas que foram

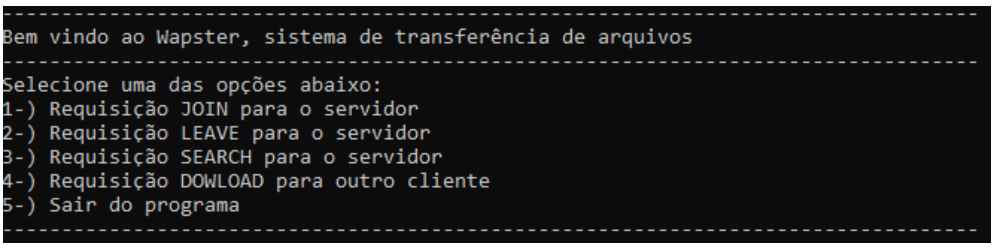
excluídas da tabela por causa da falta de ALIVE. Esse resultado é usado na hora de construir a mensagem apresentada na tela de usuário. Implementado entre as linhas 214 e 231.

- **aliveList -** Método usado pela operação ALIVE para extrair lista de peers atuais, assim a ThreadAlive sabe para quais endereços devem ser enviadas as mensagens requisitando o ALIVE_OK. Está entre as linhas 234 e 247.

4. Classe Peer

A classe Peer apresenta um grau de complexidade maior do que a classe Servidor durante sua implementação. Pois enquanto a classe Servidor só trabalhava mensagens enviadas via UDP, a classe Peer também precisa trabalhar com orientação a conexões devido ao processo de download de arquivo entre peers. Além disso, a classe Peer tem um outro complicador que é a entrada de usuário, assim temos que ter um certo controle na parte de lançamento de threads, pois agora as ações vão depender muito do usuário.

Da mesma forma que para o Servidor na seção 3, vamos começar explicando as classes aninhadas e o método Main dentro da classe Peer do arquivo Peer.java. Mas, vamos começar com a ordem inversa da última seção, pois o método Main da classe Peer possui a implementação do painel interativo para o usuário, o que é a principal fonte de eventos para geração de request. Segue uma imagem com painel interativo da aplicação e uma breve explicação de cada uma das escolhas possíveis:



Painel interativo do programa do peer

- **Opção 1 -** Realizar JOIN com o servidor. Ao escolher essa opção será perguntado ao usuário qual a pasta com os arquivos que ele deseja adicionar ao sistema. O console é travado só sai quando o usuário digita um nome de diretório válido. O caminho para o diretório deve ser absoluto. Após digitar essa informação o programa lança uma thread para envio da mensagem de JOIN para o servidor. Entre as linhas *79* e *124*.
- **Opção 2 -** Realizar LEAVE do sistema. Após escolher essa opção não é necessário digitar mais nenhuma informação no painel. O programa vai lançar uma thread que envia a mensagem de LEAVE. Detalhe, não é possível realizar LEAVE sem ter realizado JOIN. Entre as linhas *127* e *145*.

- **Opção 3 -** Realizar SEARCH para um arquivo. Quando essa opção é feita o console também pergunta para o usuário o nome do arquivo a ser buscado. Depois, é lançada uma thread que envia a mensagem para o servidor e que fica escutando a resposta sem travar o painel de usuário. Entre as linhas *148* e *179*.
- **Opção 4 -** Quando um peer quer realizar DOWNLOAD de um arquivo de outro peer. Nessa opção são solicitadas três informações do usuário: arquivo a ser baixado, ip do peer fonte e porta do peer fonte. Para usar essa opção é obrigatório que o usuário tenha feito um SEARCH para o mesmo arquivo, isso foi implementado devido ao mecanismo de retry caso o DOWNLOAD tenha sido negado pelo peer fonte. Entre as linhas *182* e *226*.
- **Opção 5 -** Escolhida quando o usuário deseja sair do programa. Caso o usuário não tenha feito o LEAVE solicita que o usuário realize antes de sair do sistema. Entre as linhas *228* e *235.*

Explicado o funcionamento do menu interativo, vamos passar para as classes aninhadas que foram implementadas dentro da classe Peer. Segue uma breve explicação de cada classe implementada.

- **ThreadEnvio -** Classe que estende thread. Seu método run é usado para tratamento de algum comando executado pelo usuário usando o painel interativo. Por exemplo, se um usuário Peer deseja fazer JOIN, ele digita o comando 1 no painel e depois a pasta com os arquivos, após isso será lançada uma ThreadEnvio para enviar a mensagem de JOIN e aguardar a resposta de JOIN_OK, assim o painel não é bloqueado. Então, resumindo, o método run dessa classe é um grande “if” que vai realizar um envio de mensagem com base no que o usuário colocar no painel interativo. Implementada entre as linhas *255* e *708*. Mais detalhes sobre cada requisição é tratada são dados mais à frente nessa mesma seção.
- **ThreadDownload -** Classe que estende thread. Quando o usuário faz o JOIN uma thread dessa classe é lançada. Ela é responsável por estabelecer um canal de escuta TCP na mesma porta registrada para aquele usuário no servidor. Quando um outro peer faz solicitação de DOWNLOAD existe um processamento para aceitar ou negar o request, verificar se o arquivo existe e enviar o arquivo em chunks via TCP. Implementado entre as linhas *711* e *810*.
- **ThreadAlive -** Mais uma classe que estende thread. Essa é a classe responsável por estabelecer um canal de escuta UDP na porta registrada no servidor. Essa porta fica escutando as mensagens ALIVE enviadas pelo servidor e devolve o ALIVE_OK para o servidor, buscando manter o peer vivo dentro do sistema. Implementado entre as linhas *813* e *881*.

Explicamos em alto nível as unidades de código envolvidas no Peer (métodos e classes). Agora vamos falar um pouco sobre qual é o fluxo da classe Peer para cada operação principal feita (JOIN, LEAVE, SEARCH, ALIVE, DOWNLOAD, UPDATE.)

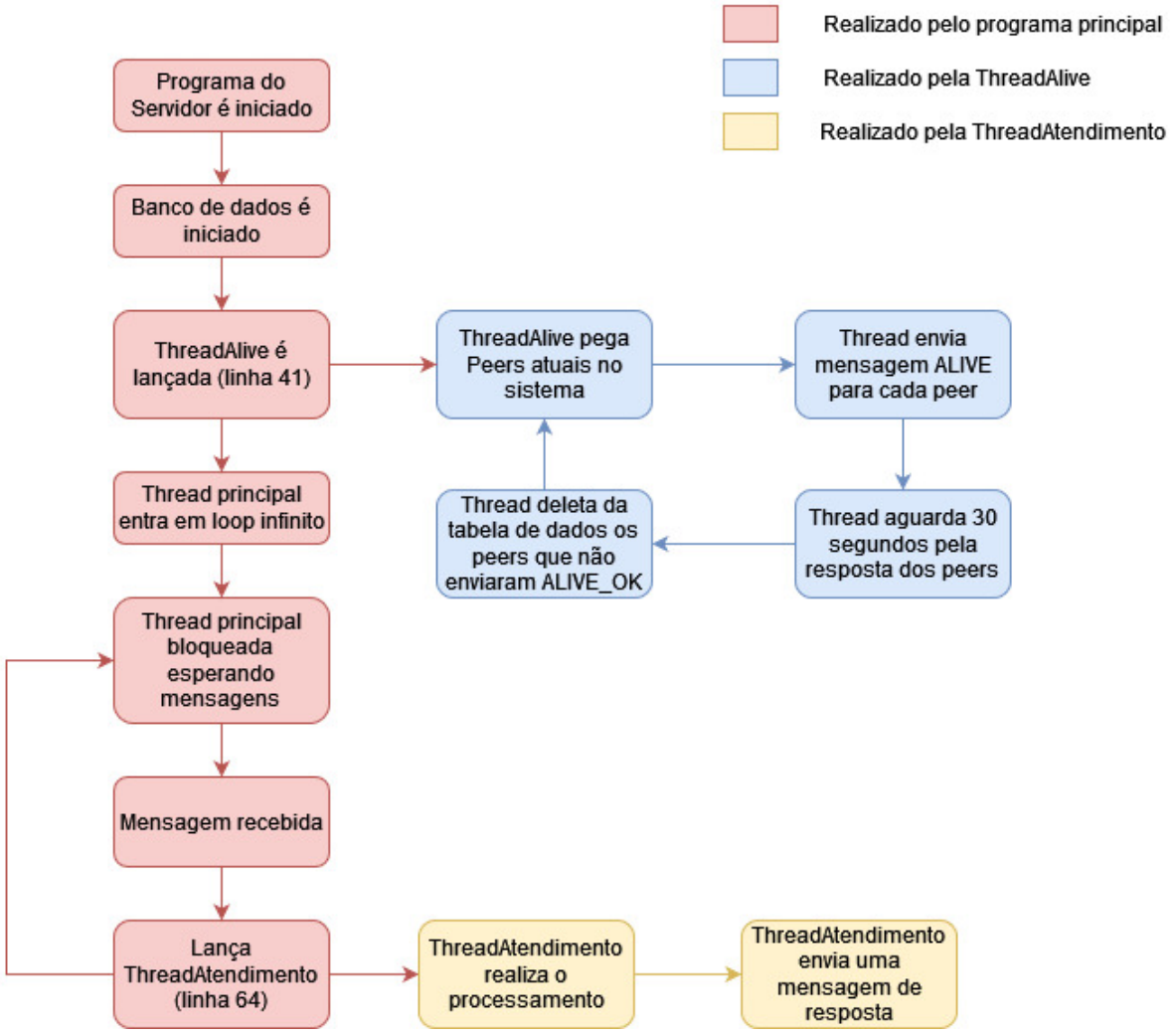
- **JOIN -** Após o usuário escolher a opção de JOIN e digitar o caminho para o diretório, uma ThreadEnvio é levantada para mandar a mensagem de JOIN para o Servidor. O conteúdo dessa mensagem é uma string com o nome de todos os arquivos da pasta de JOIN, o nome dos arquivos está separado por "//". Quando o Servidor recebe a mensagem e envia o JOIN_OK o Peer apresenta uma mensagem de sucesso para o usuário através do console. Ver linhas entre 323 e 413.
- **LEAVE -** Quando o usuário faz o LEAVE, uma ThreadEnvio é lançada e uma mensagem é enviada para o Servidor. O conteúdo da mensagem é a porta do peer que está registrada no Servidor, assim o servidor sabe quais linhas deve remover da tabela. Só lembrando que a classe Peer tem uma variável global para todas as classes aninhadas que é a portaPeer, essa variável contém o número da porta usada pelo Peer no JOIN e que está registrada no servidor como porta daquele peer. Ver linhas entre 316 e 477.
- **SEARCH -** O usuário insere o arquivo que deseja buscar. Uma mensagem é enviada para o Servidor, sendo o seu conteúdo o endereço do peer e o nome do arquivo. O Servidor faz o parse do conteúdo, realiza uma query buscando peers que contém o arquivo e devolve uma mensagem para o endereço obtido do conteúdo da primeira mensagem. O conteúdo da mensagem devolvida é uma lista de endereços de peers.
- **ALIVE -** As mensagens de ALIVE são enviadas pelo Servidor para todos os Peers de 30 em 30 segundos. Essa mensagem é enviada para a porta do peer que está registrada no servidor. Depois que um Peer faz join a porta que foi registrada no servidor é reservada para fazer a escuta das mensagens ALIVE. Outras mensagens enviadas pelo Peer para o servidor usam outras portas, mas veja que sempre enviam qual a porta identificadora do peer que está sendo usada pelo ALIVE. Quando uma mensagem de ALIVE é enviada, a ThreadAlive recebe essa mensagem e envia um ALIVE_OK para o servidor.
- **DOWNLOAD -** Quando o JOIN é realizado, a porta registrada no Servidor abre uma escuta TCP esperando requisições de outros peers diretamente para DOWNLOAD. Quando um Peer solicita DOWNLOAD para um outro Peer se estabelece uma conexão TCP e, mesmo sendo uma conexão TCP, ainda é enviado um objeto mensagem, esse objeto vai conter a informação do arquivo solicitado. O Peer solicitado pode aceitar ou negar a operação. Se aceito envia uma mensagem com valor DOWNLOAD, caso rejeitado uma mensagem de DOWNLOAD_NEGADO. Quando o DOWNLOAD é negado o peer solicitante faz um retry para algum outro peer com o arquivo, caso não esperasse um tempo e depois se faz uma nova solicitação para o mesmo peer. A parte de DOWNLOAD do peer solicitante encontra-se entre as linhas

558 e 706, na classe ThreadEnvio. Já a parte de DOWNLOAD do peer solicitado encontra-se entre as linhas 722 e 808, na classe ThreadDownload.

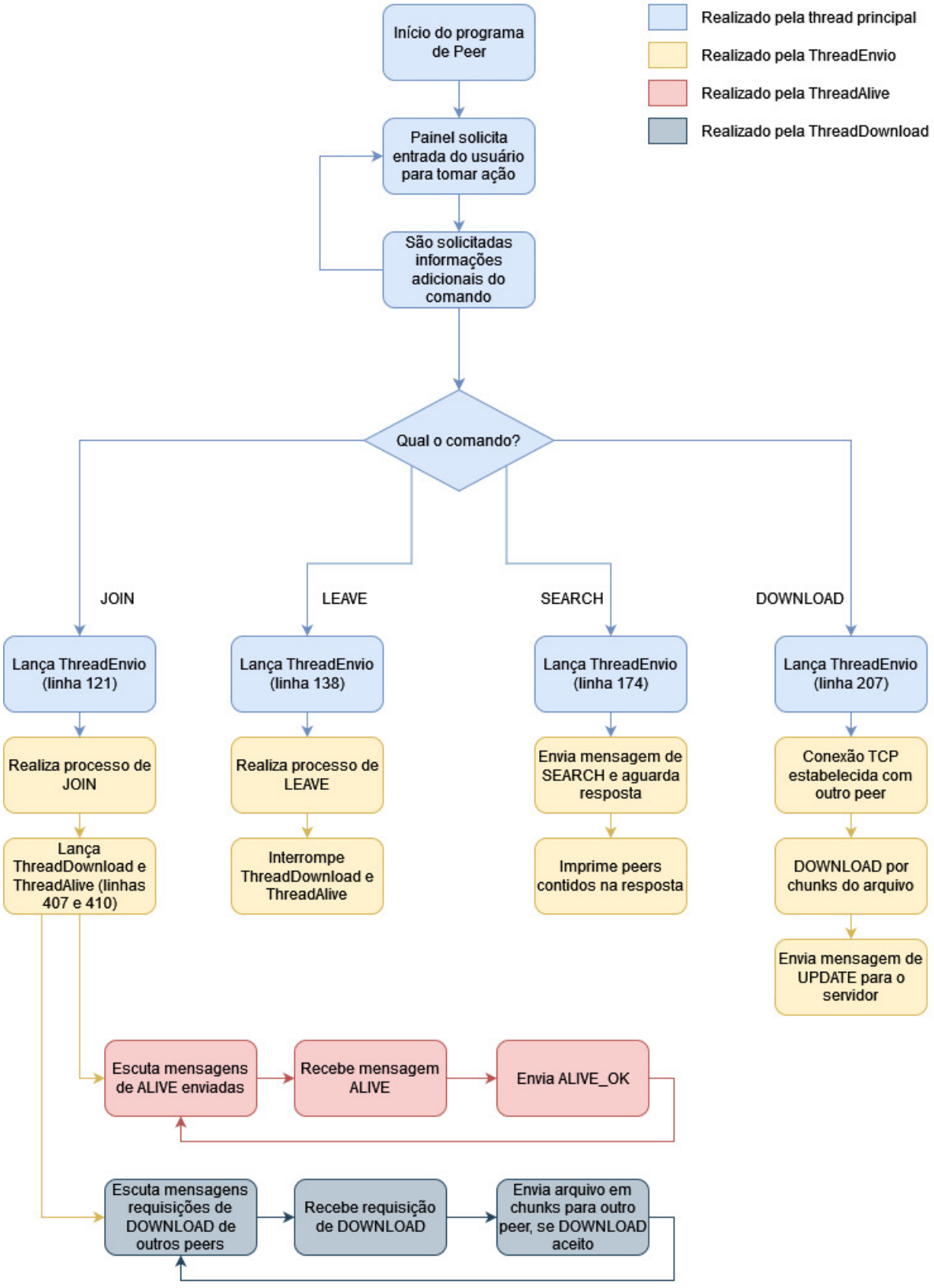
- **UPDATE -** A parte do UPDATE está muito relacionada à parte do DOWNLOAD, pois quando o DOWNLOAD de um arquivo é concluído o Peer que recebeu o arquivo envia imediatamente uma mensagem de UPDATE para o servidor. Implementação entre as linhas 624 e 654, dentro da implementação de DOWNLOAD do requerente.

5. Esquemas de Thread

Ao invés de explicar como funciona o lançamento de threads por texto, vamos usar um diagrama para cada um dos programas.



Esquema de threads do programa Servidor.java

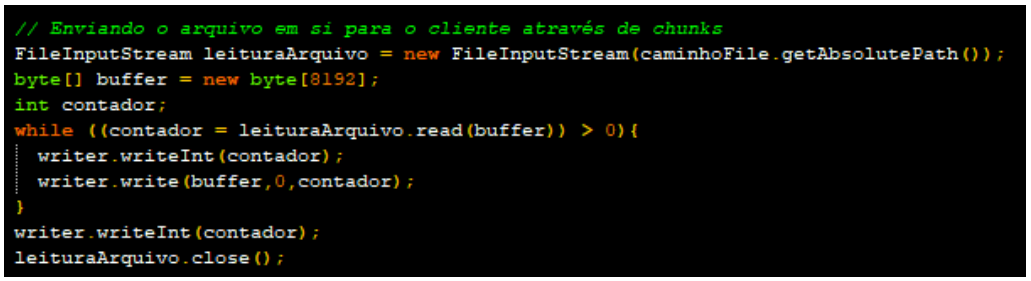


Esquema de threads do programa Peer.java

6. Transferência de arquivos gigantes

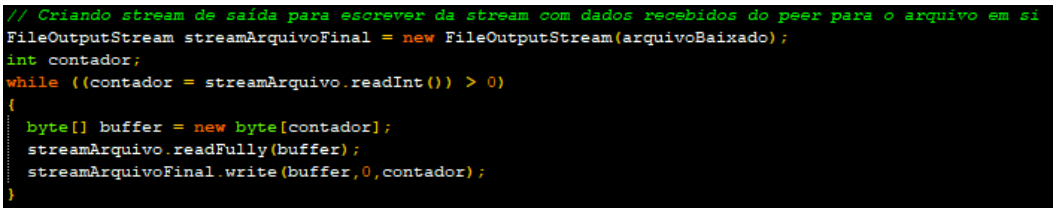
A ideia usada na transferência de arquivos gigantes foi o envio por chunks de dados com tamanho determinado, assim não seria necessário colocar todo o arquivo na memória para transferência de uma vez. A ideia em si foi até discutida em sala de aula, mas a implementação em si foi amplamente inspirada na dica da seguinte página da comunidade da Oracle: https://community.oracle.com/tech/developers/discussion/1179992/transferring-large-files. Além de dar o mapa da implementação em Java também foi possível obter dicas de quais classes usar para realizar o processo, além de observar erros de outras pessoas na hora implementação.

A implementação em si é através de um laço while que itera pelos bytes do arquivo e vai preenchendo e enviando partes do arquivo em um buffer de dados. Esse processo está amplamente sincronizado com a parte do cliente da aplicação que também vai recebendo os dados e escrevendo eles na mesma ordem. Seguem duas imagens das partes de código do projeto referentes ao servidor e cliente na transferência de arquivos, ambos no caso peers.



```
// Enviando o arquivo em si para o cliente através de chunks
FileInputStream leituraArquivo = new FileInputStream(caminhoFile.getAbsolutePath());
byte[] buffer = new byte[8192];
int contador;
while ((contador = leituraArquivo.read(buffer)) > 0){
    writer.writeInt(contador);
    writer.write(buffer,0,contador);
}
writer.writeInt(contador);
leituraArquivo.close();
```

Código da parte do servidor na ThreadDownload (linhas 772 a 781 do Peer.java)



```
// Criando stream de saída para escrever da stream com dados recebidos do peer para o arquivo em si
FileOutputStream streamArquivoFinal = new FileOutputStream(arquivoBaixado);
int contador;
while ((contador = streamArquivo.readInt()) > 0)
{
    byte[] buffer = new byte[contador];
    streamArquivo.readFully(buffer);
    streamArquivoFinal.write(buffer,0,contador);
}
```

Código da parte do cliente na ThreadEnvio (linhas 604 e 611 do Peer.java)

Mas antes de implementar essa solução foi feita uma tentativa de criar um array de bytes com o tamanho do arquivo para fazer transferência. Durante o desenvolvimento na IDE não tive nenhum problema, nenhuma exceção foi lançada e esse método mais “direto” funcionou. Porém, quando tentei executar esse código no cmd sempre gerava a exceção “java.lang.OutOfMemoryError: Java heap space” quando ia transferir um arquivo com mais de 0,5 GB. Descobri que essa exceção era justamente gerada porque essa criação de um array de bytes “enorme” excedia o espaço reservado para a JVM quando executamos um programa pelo CMD.

O problema é que como desenvolvi o projeto usando a IDE eclipse, eu acredito que a própria IDE alterava as configurações da JVM usada nas execuções pela IDE, assim a exceção de falta de memória nunca era lançada. Inicialmente o código que gerava a exceção de falta de memória foi baseado no seguinte tutorial do Youtube: https://www.youtube.com/watch ?v=GLrlwwyd1gY.

7. Manual técnico para execução do código

Essa seção final é só para facilitar a execução do programa em Java, pois os comandos necessários estão aqui, bem como a estrutura de pastas usada no vídeo.

Compilação dos códigos

- javac -encoding Latin1 -cp "./libs/*" -sourcepath ./servidor/ ./servidor/*.java
- javac -encoding Latin1 -cp "./libs/*" -sourcepath ./peer/ ./peer/*.java

Execução do servidor

- java -cp .;"./libs/*" servidor.Servidor

Execução do peer

- java -cp .;"./libs/*" peer.Peer

Estrutura de pastas

- Projeto_Final
    - bancoDados
    - libs
        - sqlite-jdbc.jar
        - json.jar
    - peer
        - Peer.java
        - Mensagem.java
    - servidor
        - Servidor.java
        - Mensagem.java

Observações fundamentais:

- A pasta bancoDados com esse nome é necessária para o projeto funcionar, recomendo usar a estrutura de pasta sugerida
- Foi necessário usar a codificação Latin1 devido aos comentários dos códigos que sempre geravam erro
- Organizei os dois .jars externos numa única pasta também por facilidade