Universidade Federal do ABC

Gabriel Zolla Juarez
RA: 11201721446

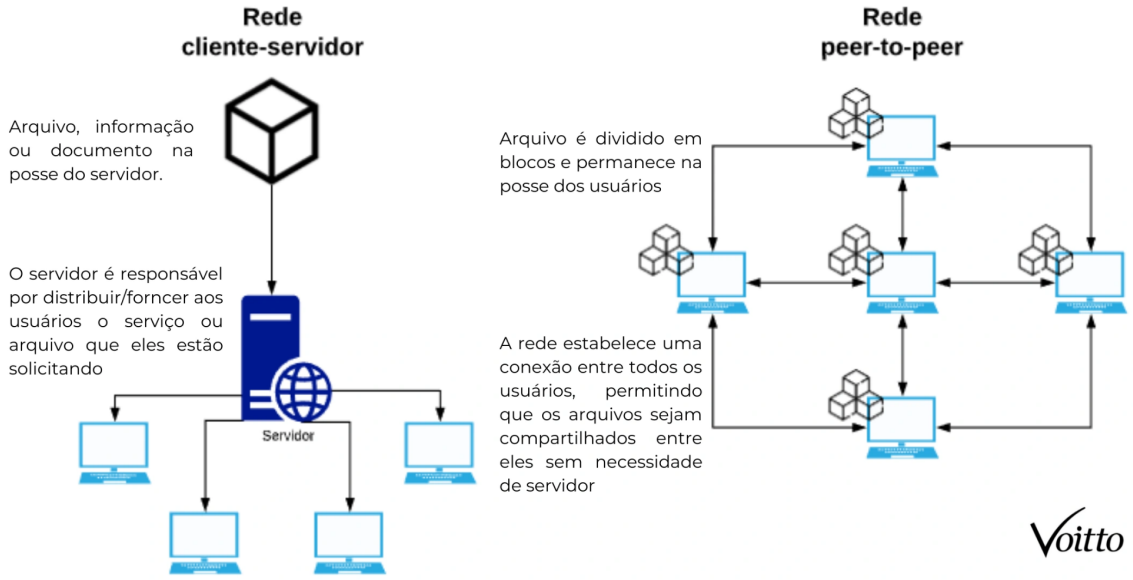
NAPSTER P2P FILE TRANSFER

Santo André
2020

1. Introdução

O sistema Napster, criado por Shawn Fanning em 1999 foi um serviço de streaming de música que revolucionou o mundo da computação, tendo em vista que fora o pioneiro a estabelecer redes de compartilhamento, onde indivíduos podiam partilhar as músicas diretamente entre usuários. Esse sistema tem como base a arquitetura P2P (peer-to-peer), que difere do sistema tradicional de um servidor centralizado, na qual todos os nós da rede (isto é, os computadores conectados à ela) funcionam tanto como servidor quanto como usuário. Esse modelo é muito presente nas aplicações de Torrent nos dias de atuais. Neste sistema, os usuários localizados na rede fornecem os arquivos que detém, e recebem o que desejam baixar. Abaixo, tem-se uma comparação dos sistemas de servidor centralizado, ao lado de uma arquitetura peer-to-peer:

Imagem 1: representação de um sistema cliente-servidor e do peer-to-peer (P2P)



Referência: https://www.voitto.com.br/blog/artigo/o-que-e-rede-p2p

Como vantagens, pode-se dizer que a arquitetura P2P fornece uma maior disponibilidade de arquivos, tendo em vista que estes não advém de um servidor centralizado, mas sim de outros usuários compartilhando arquivos. Vale ressaltar, no entanto, que existe um servidor que assume uma função de gerar as conexões entre os peers, guardar informações referente à eles, etc. Além disso, destaca-se que mesmo que ocorra uma queda na conexão de algum computador, ainda existem outros na rede disponíveis para compartilhar o arquivo, o que não aconteceria em um cenário de servidor centralizado caso o mesmo caísse.

Entretanto, também existem desvantagens. A segurança é o principal deles, uma vez que alguns nós podem incorporar malwares nos arquivos, adquirir informações do computador e do usuário, e, por isso, muitas dessas redes utilizam

sistema de reputação, comentários e alguns tipos de criptografia para identificar os peers mal-intencionados.

No projeto em questão, far-se-á um sistema semelhante ao Napster, porém com algumas reduções. Para compreender a aplicação em questão, são necessários alguns conceitos fundamentais que estão inclusos na mesma, como processos, Threads, protocolos UDP e TCP e Sockets. Portanto, discorrer-se-á sobre estes, a fim de assimilar como o sistema funciona “por baixo dos panos”.

2. Conceitos fundamentais

As aplicações desktop são baseadas em processos, isto é, uma sequência de comandos, linhas de código e entradas/saídas que ficam sendo executadas à medida que o programa está em uso. Com isso, surge o primeiro conceito importante na aplicação peer-to-peer: as Threads. Elas funcionam como um subsistema, dividindo um processo em várias tarefas, com a possibilidade de compartilhamento de recurso e informação, e, consequentemente, permitem um funcionamento paralelo de diferentes instruções de execução. No sistema em questão, serão fundamentais para a comunicação entre usuário e servidor (que armazena as informações referentes aos peers), já que o servidor recebe requisições paralelas concomitantemente, bem como servirão para manter as conexões TCP e UDP que serão vistas a seguir.

As aplicações que englobam redes de computadores são divididas em camadas, como as de rede, enlace e transporte. Esta última é fundamental na transferência de dados entre dois computadores, e será a base de toda a nossa arquitetura, já que compartilhamento de dados é o alicerce da aplicação. Para a camada de transporte, os protocolos mais utilizados são o TCP e o UDP. A função dos dois é semelhante, no entanto, existem algumas nuances entre eles. O TCP é orientado à conexão, isto é, uma conexão é estabelecida entre os nós, fornece confiabilidade, que garante a entrega de pacotes, com recuperação de pacotes perdidos ou corrompidos, e exclusão de pacotes duplicados. O UDP, por sua vez, é um protocolo mais simples e rápido, que não necessita de conexão e, com isso, não garante confiabilidade.

Deixando de lado as vantagens e desvantagens de cada um e focando em sua funcionalidade, destaca-se que ambos serão utilizados na aplicação. O TCP será fundamental para a transferência de arquivos entre os peers, tendo em vista que a confiabilidade é de extrema importância nessa funcionalidade, utilizando o conceito de Sockets para fazê-lo. As Sockets, simplificadamente, provêm a comunicação entre dois nós, sendo necessário apenas o IP e a porta dos nós. O UDP, por sua vez, será o protocolo encarregado de realizar a comunicação entre o servidor e o cliente, já que, para tal, não é necessário estabelecer conexão e nem garantir a entrega total de pacotes, e, por isso, utilizam os Datagram Packets, que são encarregados deste serviço em aplicações que não requerem conexão.

3. Sistema desenvolvido

3.1. Servidor.java

Como visto anteriormente, o servidor será a base para armazenar as informações referentes a quais arquivos os peers possuem. Primeiramente, descrever-se-á as estruturas necessárias para construir o servidor. Serão utilizados HashMaps, que basicamente são estruturas que contêm uma tupla chave-valor. Tendo em vista que existe uma concorrência na atualização de informações do servidor, serão necessários os ConcurrentHashMaps, uma instância específica dos HashMaps utilizados para aplicações que contenham tal característica. Dois HashMaps serão fundamentais para o sistema: o primeiro, denominado *filesTable* contém a tupla de String e List<String>, que correspondem ao peerID (que é uma String contendo IP:Porta), e a lista de arquivos que cada peer possui, respectivamente. O segundo, por sua vez, é o *peerRelation*, que contém a tupla Integer (que corresponde à porta UDP de comunicação entre servidor e peer, que é designada pelo sistema operacional) e String (porta utilizada na comunicação TCP). Para ilustrar melhor os atributos e seus getters e setters, vide imagem a seguir:

```
// <127.0.0.1:8080, arquivos>
public static ConcurrentHashMap<String, List<String>> filesTable;
// <PortaUDP, PortaTCP)
public static ConcurrentHashMap<Integer, String> peerRelation;

public static ConcurrentHashMap<Integer, String> getPeerRelation() {
    return peerRelation;
}

public static void setPeerRelation(ConcurrentHashMap<Integer, String> peerRelation) {
    Servidor.peerRelation = peerRelation;
}

public static void setFilesTable(ConcurrentHashMap<String, List<String>> filesTable) {
    Servidor.filesTable = filesTable;
}

public ConcurrentHashMap<String, List<String>> getFilesTable() {
    return filesTable;
}
```

Uma vez que o servidor recebe requisições concorrentes, será necessária a utilização de Threads, conceito descrito em ***2. Conceitos Fundamentais***. Portanto, nossa classe ServerThread terá extends Thread, e, com isso, deve-se ter o método run(). Nela, tratar-se-á cada ação requisitada pelo peer, tal como join, search, entre outras, através de funções externas. Além disso, será de grande utilidade a biblioteca GSON, recomendada pelo professor, para conversão de Strings para a classe Mensagem (que será descrita em breve), e vice-versa, já que a troca de mensagens UDP só pode ser feita por Strings, e não por objetos. A classe ServerThread em questão pode ser vista na imagem a seguir:

```java
public static class ServerThread extends Thread {

    private DatagramPacket recPack;
    private DatagramSocket serverSocket;

    public ServerThread(DatagramPacket recPack, DatagramSocket serverSocket) {
        this.recPack = recPack;
        this.serverSocket = serverSocket;
    }

    public void run() {
        Gson gson = new Gson();

        // Tratar a mensagem recebida
        String info = new String(recPack.getData(), recPack.getOffset(), recPack.getLength()); // Mensagem recebida
        Mensagem mensagem = new Mensagem();
        mensagem = gson.fromJson(info, Mensagem.class); // Converter a mensagem recebida (GSON) para um objeto da classe Mensagem.
        String action = mensagem.getAction().toUpperCase(); // Ação requisitada pelo peer

        try {

            // Tratar o join
            if (action.equals("JOIN")) {
                peerJoin(recPack, serverSocket, mensagem);
            }

            // Tratar o search
            else if(action.equals("SEARCH")) {
                peerSearch(recPack, serverSocket, mensagem);
            }

            // Tratar o leave
            else if(action.equals("LEAVE")) {
                peerLeave(recPack, serverSocket);
            }

            // Tratar o update
            else if(action.equals("UPDATE")) {
                peerUpdate(recPack, serverSocket, mensagem);
            }

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Vale ressaltar que os parâmetros recPack, que é o pacote que advém do peer, e o serverSocket, que, por sua vez é a Socket que abriga o servidor, serão passados na main e serão captados quando o servidor receber a mensagem do peer. Na main, ademais, serão instanciadas as fileTable e peerRelation, bem como criado o Socket do servidor, que corresponde a um DatagramSocket para a conexão UDP, e terá a porta fixa de 10098, como expresso na documentação do projeto. Para fins de demonstração do funcionamento da Hash, adicionar-se-á um print temporário: System.out.println("SITUAÇÃO DA HASH: " + filesTable.toString()), cujo intuito será mostrado posteriormente.

Para a funcionalidade JOIN, pegar-se-á o IP, porta (que será usada no TCP) e a pasta do peer através do input, bem como a porta para UDP, designada pelo SO, através do método getPort() do pacote recebido. A tupla <PortaUDP, PortaTCP> será armazenada em peerRelation, enquanto <peerID, arquivos> serão postas em filesTable, ambas pelo método .put() do HashMap. Por fim, o servidor deve responder, por meio do mesmo protocolo UDP, a String “JOIN_OK” caso a operação tenha sido bem-sucedida. Como é obrigatório a utilização da classe Mensagem para a troca de mensagens, será utilizada a biblioteca GSON, como dito anteriormente.

```java
    /*
     ** Função para tratar o JOIN do peer.
     */
    public static void peerJoin(DatagramPacket recPack, DatagramSocket serverSocket, Mensagem mensagem) throws Exception {
        Gson gson = new Gson();

        // Informações do peer para devolver o pacote
        InetAddress IPAddress = recPack.getAddress();
        int port = recPack.getPort();

        // Informações do peer (peerID == IP:Port)
        String peerID = mensagem.getIp() + ":" + mensagem.getPort();

        // Relação entre porta do SO (UDP) e porta passada pelo cliente
        peerRelation.put(port, mensagem.getPort());

        // Informações do Arquivo
        String filesString = mensagem.getMessage().trim();
        String filesArray[] = filesString.split("; "); // array de arquivos para guardar na hash.

        // Armazenar filesArray[] em uma ArrayList
        List<String> fileList;
        fileList = Arrays.asList(filesArray);

        // Armazenar fileList no HashMap referente a qual peer enviou os arquivos
        filesTable.put(peerID, fileList);

        // Resposta do servidor ao peer
        byte[] sendBuf = new byte[1024];
        Mensagem response = new Mensagem(); // Objeto response terá como action = "JOIN_OK" e message = peerID
        response.setAction("JOIN_OK");
        response.setMessage(peerID);
        String responseJSON = gson.toJson(response, Mensagem.class); // conversão do objeto da classe Mensagem para GSON.
        sendBuf = responseJSON.getBytes();
        DatagramPacket sendPacket = new DatagramPacket(sendBuf, sendBuf.length, IPAddress, port);
        serverSocket.send(sendPacket);

        // Após mensagem ser enviada
        System.out.println("Peer ["+IPAddress+"]:["+mensagem.getPort()+"] adicionado com arquivos " + fileList.toString());
    }

```

O método peerLeave, por sua vez, utilizado para remover peers da lista, será relativamente simples. Quando recebida a requisição, basta removermos o usuário e os arquivos que o mesmo possui da HashMap filesTable, bem como excluí-lo da lista peerRelation, ambos com o método .remove(), tal como é possível averiguar nas linhas 288 e 289 da imagem a seguir. Vale ressaltar que antes da remoção, deve-se percorrer peerRelation para adquirir qual a porta de comunicação de TCP do peer, para que seja possível removê-la de filesTable com sucesso. Como resposta, o servidor envia um “LEAVE_OK”. Da mesma forma, será usada a biblioteca GSON para que a comunicação seja feita a partir da classe Mensagem. O código da função peerLeave pode ser observado a fio:

```java
/*
** Função para tratar o LEAVE do peer.
*/
public static void peerLeave(DatagramPacket recPack, DatagramSocket serverSocket) throws Exception {
    Gson gson = new Gson();

    // Informações do peer (UDP)
    InetAddress IPAddress = recPack.getAddress();
    int port = recPack.getPort();

    // Port de comunicação TCP
    String tcpPort = peerRelation.get(port);
    String peerID = IPAddress.getHostAddress() + ":" + tcpPort;

    // Remove o peer da hash e os arquivos dele
    filesTable.remove(peerID);
    peerRelation.remove(port);

    // Gerar um objeto Mensagem para comunicação
    Mensagem response = new Mensagem(); // Objeto response terá como action = "JOIN_OK" e message = peerID
    response.setMessage("LEAVE_OK");
    String responseJSON = gson.toJson(response, Mensagem.class); // conversão do objeto da classe Mensagem para GSON.

    // Resposta do servidor ao peer
    byte[] sendBuf = new byte[1024];
    sendBuf = (responseJSON).getBytes();
    DatagramPacket sendPacket = new DatagramPacket(sendBuf, sendBuf.length, IPAddress, port);
    serverSocket.send(sendPacket);

    // Após mensagem ser enviada
    System.out.println("Mensagem enviada...");
}
```

Para o search, é necessário devolver todos os peers que contém determinado arquivo. Para tal, será preciso percorrer todas as tuplas da tabela filesTable e consultar todos os índices da List<String>, que corresponde ao valor da tupla chave-valor do HashMap. A fim de ilustrar como funciona essa execução, segue o trecho de código:

```java
public static void peerSearch(DatagramPacket recPack, DatagramSocket serverSocket, Mensagem mensagem) throws Exception {
    Gson gson = new Gson();

    // Informações do peer
    InetAddress IPAddress = recPack.getAddress();
    int port = recPack.getPort();

    // Port de comunicação TCP
    String tcpPort = peerRelation.get(port);
    String peerID = IPAddress.getHostAddress() + ":" + tcpPort;

    String fileDesired = mensagem.getMessage().trim(); // Arquivo desejado pelo peer
    ArrayList<String> arrayOfPeers = new ArrayList<>(); // Lista temporária para armazenar os peers que contém o arquivo

    for(Map.Entry<String, List<String>> entry : filesTable.entrySet()) {
        if(entry.getValue().contains(fileDesired)) {
            arrayOfPeers.add(entry.getKey());
        }
    }

    // Gerar um objeto Mensagem para comunicação
    Mensagem response = new Mensagem(); // Objeto response terá como action = "JOIN_OK" e message = peerID
    response.setMessage("Peers com arquivo solicitado: " + arrayOfPeers.toString());
    String responseJSON = gson.toJson(response, Mensagem.class); // conversão do objeto da classe Mensagem para GSON.


    // Resposta do servidor ao peer
    byte[] sendBuf = new byte[1024];
    sendBuf = (responseJSON).getBytes();
    DatagramPacket sendPacket = new DatagramPacket(sendBuf, sendBuf.length, IPAddress, port);
    serverSocket.send(sendPacket);
}
```

Como se pode ver, a String fileDesired (linha 318) contém o arquivo que o peer deseja. Far-se-á necessária a utilização de uma lista temporária denominada arrayOfPeers (linha 319) que será composta por todos os peers que contém o arquivo desejado. Nas linhas 321 a 325, o loop for percorre o HashMap, e a variável 'entry' é cada tupla percorrida. Portanto, basta verificarmos se entry.getValue() contém o arquivo desejado. Se sim, armazenamos no arrayOfPeers. Retornamos, por fim, este array através da classe Mensagem.

Por fim, tem-se o peerUpdate, que serve para atualizar a situação do peer quando o mesmo baixou um novo arquivo. O código em si é muito semelhante aos demais, porém com uma pequena nuance:

```
340     public static void peerUpdate(DatagramPacket recPack, DatagramSocket serverSocket, Mensagem mensagem) throws Exception {
341         Gson gson = new Gson();
342 
343         // Informações do peer
344         InetAddress IPAddress = recPack.getAddress();
345         int port = recPack.getPort();
346         String fileDownloaded = mensagem.getMessage().trim();
347 
348         // Port de comunicação TCP
349         String tcpPort = peerRelation.get(port);
350         String peerID = IPAddress.getHostAddress() + ":" + tcpPort;
351 
352         List<String> newList = new ArrayList<>();
353         newList.addAll(filesTable.get(peerID));
354         newList.add(fileDownloaded);
355 
356         filesTable.remove(peerID);
357         filesTable.put(peerID, newList);
358 
359 
360         // Gerar um objeto Mensagem para comunicação
361         Mensagem response = new Mensagem(); // Objeto response terá como action = "JOIN_OK" e message = peerID
362         response.setMessage("UPDATE_OK");
363         String responseJSON = gson.toJson(response, Mensagem.class); // conversão do objeto da classe Mensagem para GSON.
364 
365 
366         // Resposta do servidor ao peer
367         byte[] sendBuf = new byte[1024];
368         sendBuf = responseJSON.getBytes();
369         DatagramPacket sendPacket = new DatagramPacket(sendBuf, sendBuf.length, IPAddress, port);
370         serverSocket.send(sendPacket);
371 
372     }
373 }
```

Objetos da classe List<> são imutáveis após instanciá-los, ou seja, não podemos remover ou adicionar objetos fora de onde ela foi instanciada, caso contrário recebemos um erro UnsupportedOperationException. Portanto, uma pequena manipulação é necessária: a criação de uma nova List<> denominada newList (linha 352), na qual adicionaremos todos os elementos de filesTable.get(peerID), isto é, todos os arquivos pertencentes à peerID (linha 353) somado ao novo arquivo baixado (linha 354). Então, remover-se-á a tupla de peerID e adicionará uma nova tupla contendo <peerID, newList>, como visto na linha 357. Por fim, basta enviar uma mensagem ao cliente, através de UDP, um objeto da classe Mensagem cujo atributo message será "UPDATE_OK".

Por fim, tem-se a classe KeepAliveThread, que também funciona como uma Thread já que executa concomitantemente às requisições vindas do peer. Essa função basicamente envia a cada 30 segundos, para todos os peers, de forma a confirmar se estes ainda estão vivos no sistema, isto é, se estão ativos, enviando uma mensagem por UDP com a string “ALIVE”. Caso não receba uma resposta “ALIVE_OK”, o servidor irá remover todas as informações do peer da HashMap (seu peerID e arquivos). Vale ressaltar que desta vez, o servidor que começa enviando a mensagem, enquanto o peer recebe-a e responde, o que gera uma pequena alteração no momento de criar a Socket, já que não é preciso passar a porta para o server, mas apenas ao peer. Um print da classe se encontra abaixo:

```
151    public static class KeepAliveThread extends Thread{
152        private DatagramSocket serverSocket;
153
154        public KeepAliveThread(DatagramSocket serverSocket) {
155            this.serverSocket = serverSocket;
156        }
157
158        public void run() {
159            while(true) { //loop para executar uma rodada de alive
160                try {
161                    Thread.sleep(30000);
162                    String sentence = "ALIVE";
163                    byte[] sendData = new byte[sentence.length()];
164                    sendData = sentence.getBytes();
165
166                    if (!filesTable.isEmpty()) {
167                        for (Map.Entry<String, List<String>> entry : filesTable.entrySet()) {
168                            InetAddress IPAddress = InetAddress.getByName(entry.getKey().split(":")[0]);
169                            String ip = IPAddress.toString().split("/")[1];
170                            int port = Integer.parseInt(entry.getKey().split(":")[1]);
171                            String peerID = IPAddress + ":" + port;
172
173                            DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, IPAddress, port);
174                            serverSocket.send(sendPacket);
175                            try {
176                                serverSocket.setSoTimeout(3000);
177                                byte[] data = new byte[1000];
178                                DatagramPacket recPack = new DatagramPacket(data, data.length);
179
180                                try {
181                                    serverSocket.receive(recPack);
182                                    String response = new String(recPack.getData(), 0, recPack.getLength());
183
184                                } catch (SocketTimeoutException e) {
185                                    System.out.println("Peer [" + ip + "]:[" + port
186                                            + "] morto. Eliminando seus arquivos [" + entry.getValue() + "]");
187                                    filesTable.remove(peerID);
188                                }
189                            } catch (Exception e) {
190                                e.printStackTrace();
191                            }
192                        }
193                    }
194                }catch(Exception e) {e.printStackTrace();}
195            }
```

Destacar-se-á que essa funcionalidade gerou certos problemas na hora da interação entre o peer e o servidor, pois atrapalhava a leitura das requisições advindas pelo peer, erro que não encontrem a origem. Portanto, deixarei essa funcionalidade, por ora, fora do nosso sistema, não invocando sua Thread na main.

3.2. Mensagem.java

A classe mensagem é relativamente simples. Ela funciona apenas como um armazenador de Strings que contém a requisição feita pelo e a mensagem passada pelo peer ao servidor e vice-versa. Como utilizamos a biblioteca GSON, por recomendação da documentação do projeto, converteremos constantemente de Mensagem para GSON e de GSON para Mensagem, uma vez que a transmissão de pacotes nos protocolos em questão se dá por bytes (strings) e não por classes.

```java
package napsterTransfer;

public class Mensagem {
    private String action;
    private String message;
    private String ip;
    private String port;

    public Mensagem(String action, String message) {
        this.action = action;
        this.message = message;
        this.ip = "";
        this.port = "";
    }

    public Mensagem(String action, String ip, String port, String message) {
        this.action = action;
        this.message = message;
        this.ip = ip;
        this.port = port;
    }

```

Para reduzir a quantidade de uso do método .split(), que será bastante utilizado no peer para manipular as Strings recebidas, criar-se-á quatro atributos na classe mensagem: action, que contém a requisição, message, que contém informações como um todo, ip e porta (utilizados apenas no join). Além do trecho de código acima, tem-se apenas getters e setters.

3.3. Peer.java

O código do Peer é o mais complexo da aplicação, tendo em vista que deve se comunicar tanto com o servidor - a partir do protocolo UDP - quanto com outros peers que requisitam arquivo - pelo TCP.

Primeiramente, discorrer-se-á sobre o tratamento do UDP no peer. Tendo em vista que a comunicação é concorrente, será necessário o uso de Threads, tal como apontado anteriormente. A classe em questão captura inputs do usuário contendo requisições e informações necessárias para o mesmo, que serão passadas para a variável actionString (linha 146). Dentro do método, tratar-se-á os casos específicos de download, para encaminhar para o TCP e permitir a transferência de arquivos, tal como observa-se nas linhas 148 até 151, e as demais requisições são passadas para a função requestAction, criada para tornar a classe mais enxuta e conseguir reaproveitar código para as demais requisições.

```
131  public static class peerUDPClass extends Thread {
132
133      public String peerID;
134      public DatagramSocket peerSocket;
135
136      public peerUDPClass(String peerID, DatagramSocket peerSocket) {
137          this.peerID = peerID;
138          this.peerSocket = peerSocket;
139      }
140
141      public void run() {
142          Scanner scan = new Scanner(System.in);
143          System.out.println("Digite a ação, seguida de ':' (dois pontos) e as informações necessárias: ");
144
145          while(scan.hasNextLine()) {
146              String actionString = scan.nextLine();
147
148              if(actionString.split(": ")[0].toUpperCase().equals("DOWNLOAD")) {
149                  downloadClass dc = new downloadClass(actionString, peerID, peerSocket);
150                  dc.start();
151              }
152
153              String info;
154
155              try {
156                  info = requestAction(actionString, peerSocket, peerID);
157                  if(info.equals("LEAVE_OK")) System.exit(0);
158              } catch (Exception e) {
159                  e.printStackTrace();
160              }
161          }
162          System.out.println("Digite a ação, seguida de ':' (dois pontos) e as informações necessárias: ");
163          
164      }
165  }
```

A função requestAction referenciada acima recebe três parâmetros: a String actionString, o DatagramSocket do peer para a comunicação UDP, e a String peerID que contém uma string IP:Porta. Nessa função, será necessário utilizar diversos .split(), que é uma função que armazena palavras separadas por um determinado caracter em forma de array, de forma a adquirir e tratar as strings recebidas de forma a manipular o HashMap. O código pode ser observado a seguir:

```java
public static String requestAction(String actionString, DatagramSocket peerSocket, String peerID) throws Exception {
    int serverPort = 10098;
    Gson gson = new Gson();

    // Separar a ação (join) do resto da mensagem
    String infoArray[] = actionString.split(": ");
    Mensagem mensagem = new Mensagem();

    // Peer IP
    InetAddress IPAddress = InetAddress.getByName(peerID.split(":")[0]);

    // Tratar o leave
    if(infoArray[0].trim().toUpperCase().equals("LEAVE")) {
        mensagem.setAction("LEAVE");
    }

    // Tratar as demais mensagens
    else {
        mensagem.setAction(infoArray[0].trim().toUpperCase());
        mensagem.setMessage(infoArray[1].trim());
    }

    // Converter o objeto em GSON para enviar para o servidor
    String mensagemJSON = gson.toJson(mensagem);

    // Pacote enviado para o servidor
    byte[] sendData = new byte[1024];
    sendData = mensagemJSON.getBytes();
    DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, IPAddress, serverPort);
    peerSocket.send(sendPacket);

    // Pacote de recebimento (recebido pelo peer, vindo do servidor)
    byte[] recBuffer = new byte[1024];
    DatagramPacket recPack = new DatagramPacket(recBuffer, recBuffer.length);
    peerSocket.receive(recPack);
    String info = new String(recPack.getData(), recPack.getOffset(), recPack.getLength());

    // Converter de GSON para classe Mensagem
    Mensagem responseMessage = gson.fromJson(info, Mensagem.class);
    System.out.println(responseMessage.getMessage() + "\n");

    return info;
}
```

Como observado nas linhas de 383 a 391, a função manipula um objeto mensagem, converte para JSON, e se comunica com o servidor. No servidor, então, será tratada essa mensagem, alterando a tabela, portanto.

Antes de mostrar a função que trata o join, que é uma função relativamente complexa, observar-se-á a main, tendo em vista que é ela que inicia todas as Threads e chama o seu método run(). Quando o usuário inicia o programa, ele recebe uma mensagem de inicialização, na qual passará IP, porta e a pasta a partir do input que será armazenado na String iniciateString, como visto na linha 256 da figura abaixo, e armazenaremos peerID e peerPort para passar tais informações para o servidor.

```java
246     public static void main(String[] args) throws Exception {
247         Scanner scan = new Scanner(System.in);
248         DatagramSocket peerSocket = new DatagramSocket();
249 
250         // FORMATO: 'ação', 'pasta dos arquivos', 'etc..'
251         // E.G: join, IP, porta, C:/Desktop/pasta
252         System.out.println("Oi peer! Para inicializar, digite join: IP, PORTA, PASTA\n" +
253         "E.G: join: 127.0.0.1, 8080, C:/Desktop/pasta");
254 
255         // String de inicialização
256         String iniciateString = scan.nextLine().trim();
257 
258         // Captura da pasta para guardar no objeto
259         peerFolder = iniciateString.split(": ")[1].split(", ")[2];
260         int peerPortTCP = Integer.parseInt(iniciateString.split(": ")[1].split(", ")[1]);
261 
262         // Conexão com o servidor (UDP)
263         String peerID = joinServer(iniciateString, peerSocket);
264 
265         // Guardar IP e porta na classe peer
266         peerIP = InetAddress.getByName(peerID.split(":")[0]);
267         // peerPort = Integer.parseInt(peerID.split(":")[1]);
268         peerPort = peerPortTCP;
269 
270         // DatagramSocket keepAliveSocket = new DatagramSocket(peerPort);
271 
272         // Continuos UDP
273         peerUDPClass peerUDP = new peerUDPClass(peerID, peerSocket);
274         peerUDP.start();
275 
276         // Continuous Alive
277         // respondAlive keepAlive = new respondAlive(keepAliveSocket, peerIP);
278         // keepAlive.start();
279 
280         // Continuous TCP
281         ServerSocket serverSocket = new ServerSocket(peerPort);
282         peerTCPClass peerTCP = new peerTCPClass(serverSocket);
283         peerTCP.start();
284 
285     }
```

Nas linhas de 273 a 284, iniciamos criamos um objeto das classes peerUDPClass, peerTCPClass e seus respectivos parâmetros, e iniciamos suas Threads, que rodam, portanto, continuamente. Vale ressaltar que, pelo motivo descrito em **3.1. Servidor.java,** por uma inconveniência na execução do console, o keepAlive apesar de estar implementado, não será incluído na execução final, deixando-o apenas comentado. Segue a classe respondAlive, que apesar de não ser utilizada, está implementada corretamente, só não interage bem com a forma com que o console está sendo utilizado:

```java
public static class respondAlive extends Thread {
    public DatagramSocket peerSocket;
    public InetAddress IPAddress;

    public respondAlive(DatagramSocket peerSocket, InetAddress IPAddress) {
        this.peerSocket = peerSocket;
        this.IPAddress = IPAddress;
    }

    public void run() {

        while(true) {
            try {

                // Recebe o pacote para o Servidor
                byte[] receiveData = new byte[4096];
                DatagramPacket recPack= new DatagramPacket(receiveData, receiveData.length);
                peerSocket.receive(recPack);
                String sentence = new String(recPack.getData(), 0, recPack.getLength());
                int serverPort = recPack.getPort();
                Scanner scan = new Scanner(System.in);
                System.out.println(sentence);

                // Capta o ALIVE_OK
                String response = scan.nextLine();

                // Responde para o server
                byte[] sendData = new byte[4096];
                sendData = response.toUpperCase().trim().getBytes();
                DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, IPAddress, serverPort);
                peerSocket.send(sendPacket);

            } catch(Exception e) {e.printStackTrace();}
        }
    }
}
```

Com isso, podemos analisar o código do joinServer:

```java
public static String joinServer(String iniciateString, DatagramSocket peerSocket) throws Exception {
    int serverPort = 10098;
    Gson gson = new Gson();

    // Separar a ação (join) do resto da mensagem
    String infoArray[] = iniciateString.split(": ");

    // Separar as informações passadas por vírgulas (ip, port)
    String peerInfo[] = infoArray[1].split(", ");

    // IP e porta passados pelo peer
    InetAddress IPAddress = InetAddress.getByName(peerInfo[0].trim());
    int port = Integer.parseInt(peerInfo[1].trim());

    // Declarar mensagem
    Mensagem mensagem = new Mensagem();

    // Caso exclusivo de join (passar os arquivos na mensagem)
    // infoArray[0] = action (JOIN) && peerInfo[] = ip, port, mensagem
    mensagem.setAction(infoArray[0].trim());
    mensagem.setIp(peerInfo[0].trim());
    mensagem.setPort(peerInfo[1].trim());

    // Arquivos a partir da pasta
    String filesPath = peerInfo[2].trim();
    mensagem.setMessage(readFiles(filesPath).trim());

    // Converter o objeto em GSON para enviar para o servidor
    String mensagemJSON = gson.toJson(mensagem);

    // Pacote enviado para o servidor
    byte[] sendData = new byte[1024];
    sendData = mensagemJSON.getBytes();
    DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, IPAddress, serverPort);
    peerSocket.send(sendPacket);

    // Pacote de recebimento (recebido pelo peer, vindo do servidor)
    byte[] recBuffer = new byte[1024];
    DatagramPacket recPack = new DatagramPacket(recBuffer, recBuffer.length);
    peerSocket.receive(recPack);

    // Print resposta do server
    String info = new String(recPack.getData(), recPack.getOffset(), recPack.getLength());
    Mensagem response = gson.fromJson(info, Mensagem.class);
    System.out.println(response.getAction() + ". Sou peer " + response.getMessage() + " com arquivos: " + mensagem.getMessage());

    // Retorna informações sobre o peer
    return response.getMessage();
}
```

Tal como as demais comunicações UDP, das linhas 357 a 371 tem-se os comandos padrão de armazenar um buffer para enviar o pacote, e recebê-lo à partir do recPack. Acima, manipula-se o objeto mensagem, da classe Mensagem, para armazenar os parâmetros de ação, porta e IP. Para isso, utilizar-se-á a biblioteca GSON, novamente, para converter objeto para String e vice-versa. Além disso, armazena-se o filesPath passado por input em uma String e rodamos a função readFiles(), vista em seguida que, ao passar como parâmetro a pasta, retorna todos os arquivos presentes nela separados por ponto e vírgula, como pode ser observado na imagem a seguir:

```
418     // Função para ler todos os arquivos da pasta do peer e armazenar em uma String
419     // Referência: https://stackoverflow.com/questions/1844688/how-to-read-all-files-in-a-folder-from-java
420     public static String readFiles(String path) {
421         File folder = new File(path);
422         File[] listOfFiles = folder.listFiles();
423         String fileListString = "";
424 
425         if (listOfFiles != null) {
426             for (File file : listOfFiles) {
427                 if (file.isFile()) {
428                     fileListString += file.getName() + "; ";
429                 }
430             }
431         }
432         return fileListString;
433     }
434 }
```

Como visto anteriormente, para o download, deve-se ter uma Thread ouvindo para toda a requisição de download feita:

```
public static class downloadClass extends Thread {

    public String request;
    public DatagramSocket peerSocket;
    public String peerID;

    public downloadClass(String request, String peerID, DatagramSocket peerSocket) {
        this.request = request;
        this.peerID = peerID;
        this.peerSocket = peerSocket;
    }

    public void run() {
        // Informações sobre o arquivo e o host escolhido
        String downloadInfo[] = request.split(": ")[1].split(", ");
        String downloadIP = (downloadInfo[0]);
        int downloadPort = Integer.parseInt(downloadInfo[1]);
        String downloadFile = downloadInfo[2];

        // Gerar novamente o UDP.
        ClientPeer cp = new ClientPeer(downloadIP, downloadPort, downloadFile, peerID, peerSocket);
        cp.start();

        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

A string downloadIP será o IP passado pelo o usuário, bem como downloadPort será a porta e downloadFile o arquivo desejado. Com isso, cria-se um novo clientPeer, que será o peer que receberá o arquivo em questão.

Para compreender o clientPeer, será interessante primeiramente analisar a classe peerTCP - que será o peer que fornece o arquivo ao clientPeer - que, por ter sido instanciado e iniciada na main, o método serverSocket.accept() fica em espera, bloqueante, até uma conexão de outro peer.

```
171     public peerTCPClass(ServerSocket serverSocket) {
172         this.serverSocket = serverSocket;
173     }
174
175     public void run() {
176
177         while(true) {
178             try {
179                 // System.out.println("DENTRO DO TCP");
180                 Socket clientSocket = serverSocket.accept();
181
182                 System.out.println("Conexão requerida de " + clientSocket.getInetAddress() + ":" + clientSocket.getPort());
183
184                 // Ler do cliente
185                 BufferedReader br = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
186                 String received = br.readLine();
187
188                 // Pasta do cliente
189                 String file = received.split(",")[1].trim();
190                 String folder = received.split(",")[0].trim();
191
192                 // System.out.println("Arquivo: " + file + " na pasta peerFolder (server): " + peerFolder);
193                 File newFile = new File(peerFolder + "/" + file);
194
195                 // Enviar para o cliente
196                 OutputStream os= clientSocket.getOutputStream();
197                 DataOutputStream output = new DataOutputStream(os);
198
199                 if(Math.random() > 0.80) {
200                     output.writeBytes("DOWNLOAD_NEGADO\n");
201                     return;
202                 }
203
204                 else output.writeBytes(peerFolder + "\n");
205
206                 DataInputStream dis = new DataInputStream(clientSocket.getInputStream());
207                 FileOutputStream fos = new FileOutputStream(folder + "/" + file);
208                 byte[] buffer = new byte[4096];
209
210                 int read;
211                 while ((read=dis.read(buffer)) > 0) {
212                         fos.write(buffer,0,read);
213                 }
214
215                 System.out.println("Download concluído!");
216
217                 fos.close();
218                 dis.close();
219
```

Quando o ClientPeer é instanciado e iniciado, é então que a linha 180, (Socket clientSocket = serverSocket.accept()); é desbloqueada e então inicia-se o código. Primeiramente, o peerTCPClass lê uma mensagem do cliente contendo as informações passadas por input: ip/porta do host e arquivo desejado (e consequentemente, a pasta onde ele quer salvar o arquivo). Com isso, é possível ler os bytes do arquivo pedido a partir das estruturas de DataInputStream e FileOutputStream e escrevê-los em um arquivo para a transferência. Essas estruturas serão descritas de forma mais detalhada em **3.4. Transferência de arquivos gigantes**

Vale ressaltar que, por convenção, coloquei 80% de chance do peer aceitar a requisição de download, como visto nas linhas 199 a 201.

Um código semelhante se encontra na imagem a seguir, referente ao ClientPeer.

```
24      public ClientPeer(String hostIP, int hostPort, String file, String peerID, DatagramSocket peerSocket) {
25          this.hostIP = hostIP;
26          this.hostPort = hostPort;
27          this.file = file;
28          this.peerID = peerID;
29          this.peerSocket = peerSocket;
30      }
31
32      public void run() {
33          try {
34              s = new Socket(hostIP, hostPort);
35
36              // Enviar o arquivo desejado para o server.
37              OutputStream os= s.getOutputStream();
38              DataOutputStream serverWriter = new DataOutputStream(os);
39              // serverWriter.writeBytes(peerFolder + ", " + file + "\n");
40              serverWriter.writeBytes(peerFolder + ", " + file + "\n");
41
42              // Resposta do server
43              InputStreamReader isrServer = new InputStreamReader(s.getInputStream());
44              BufferedReader serverReader = new BufferedReader(isrServer);
45              String folder = serverReader.readLine();
46
47              // Tratar download negado
48              if(folder.equals("DOWNLOAD_NEGADO")) {
49                  System.out.println("Download negado. Tente novamente com outro peer");
50                  peerUDPClass peerUDP = new peerUDPClass(peerID, peerSocket);
51                  peerUDP.start();
52                  return;
53              }
54
55              // Estruturas para armazenar o file
56              DataOutputStream dos = new DataOutputStream(s.getOutputStream());
57              FileInputStream fis = new FileInputStream(folder + "/" + file);
58              byte[] buffer = new byte[4096];
59
60              // Leitura do arquivo
61              int read;
62              while ((read=fis.read(buffer)) > 0) {
63                  dos.write(buffer,0,read);
64              }
65
66              dos.flush();
67              fis.close();
68              dos.close();
69              serverWriter.close();
70
71              System.out.println("Arquivo " + file + " baixado com sucesso na pasta " + peerFolder);
72              s.close();
73
74              peerUDPClass peerUDP = new peerUDPClass(peerID, peerSocket);
75              peerUDP.start();
```

Após ser chamado, o método do ClientPeer cria um socket com o IP e porta do host, que será a forma de comunicação entre os dois peers. Em seguida, envia o arquivo desejado por TCP, e recebe a resposta com a pasta em que se encontra. Com isso, basta gerar as estruturas de DataOutputStream e FileInputStream, tal como visto anteriormente, armazenar um buffer para a leitura do arquivo, e impressão do mesmo na pasta desejada. Vale ressaltar que ambos devem ser Threads pois pode haver um caso de concorrência de download, onde dois peers requisitam o mesmo arquivo, por exemplo.

3.4. Transferência de arquivos gigantes

A transferência de arquivos gigantes, cujo tamanho é maior de 1GB, são possíveis justamente por conta dos trechos a seguir, que se encontram no ClientPeer e no peerTCPClass, respectivamente:

```
55                  // Estruturas para armazenar o file
56                  DataOutputStream dos = new DataOutputStream(s.getOutputStream());
57                  FileInputStream fis = new FileInputStream(folder + "/" + file);
58                  byte[] buffer = new byte[4096];
59
60                  // Leitura do arquivo
61                  int read;
62                  while ((read=fis.read(buffer)) > 0) {
63                      dos.write(buffer,0,read);
64                  }
```

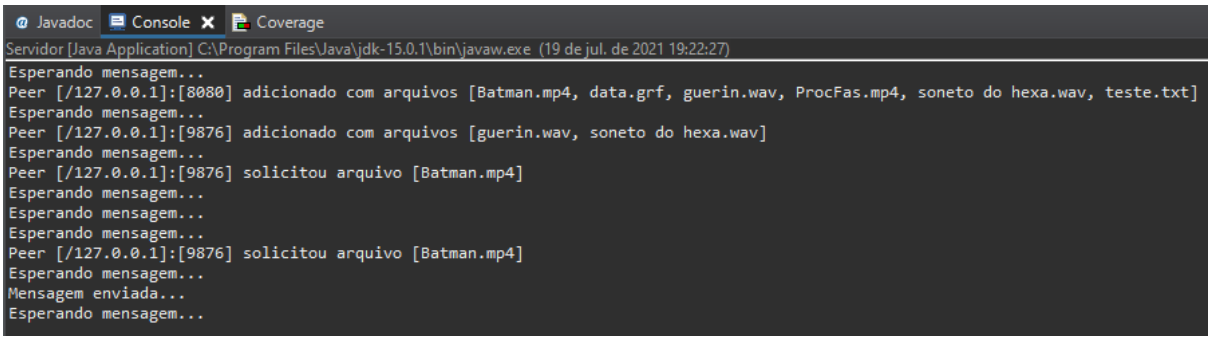
```
205
206                 DataInputStream dis = new DataInputStream(clientSocket.getInputStream());
207                 FileOutputStream fos = new FileOutputStream(folder + "/" + file);
208                 byte[] buffer = new byte[4096];
209
210                 int read;
211                 while ((read=dis.read(buffer)) > 0) {
212                         fos.write(buffer,0,read);
213                 }
214
```

Ambos os trechos permitem a transferência de um array de bytes, isto é, as variáveis ‘buffer’, que vão sendo lidas/escritas gradativamente, até o momento em que se encontrarem vazias, o que exime a responsabilidade de passar um tamanho de arquivo como parâmetro, tal como na referência [1].

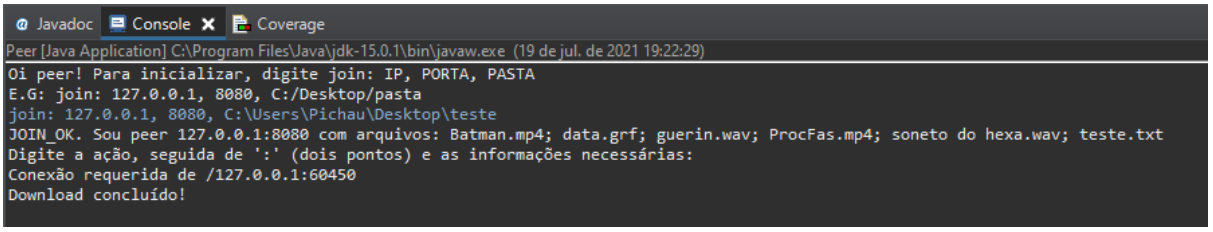
Com isso, no peerTCPClass, onde se encontra o FileInputStream, que obtém os bytes a partir de um arquivo do sistema, e será passado pelo socket através do DataInputStream, que consiste no fluxo de entrada de dados na socket. Enquanto isso, no ClientPeer, tem-se o FileOutputStream, que permite escrever os bytes em um arquivo, enquanto o DataOutputStream servirá para receber estes bytes através da socket, funcionando como um fluxo de saída.

4. Exemplo de uso

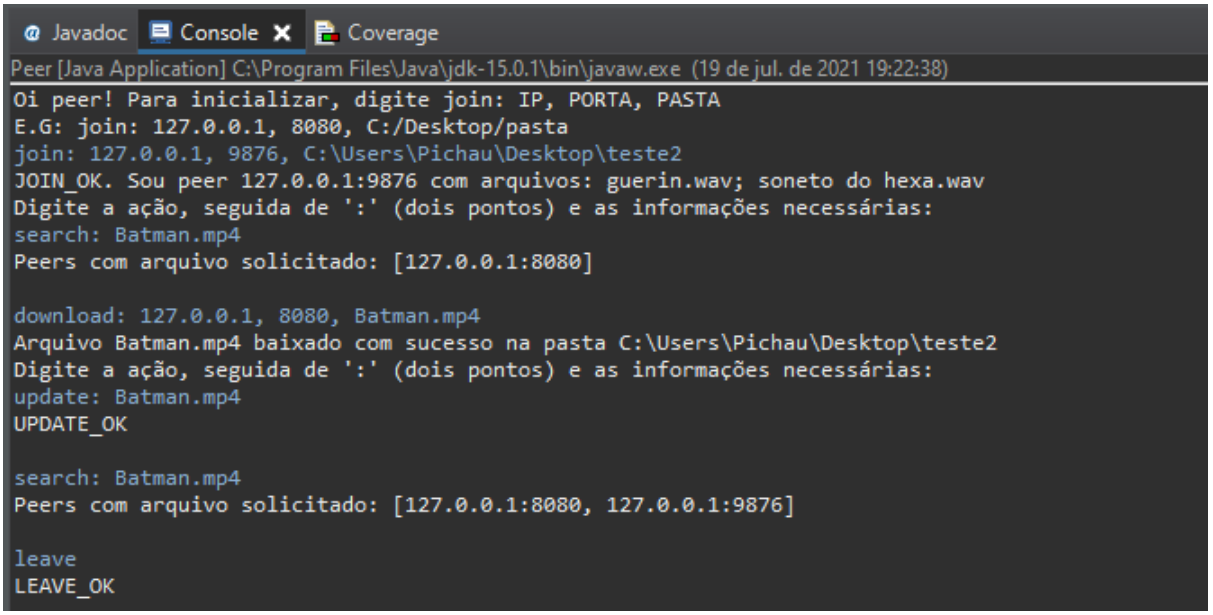
Server:



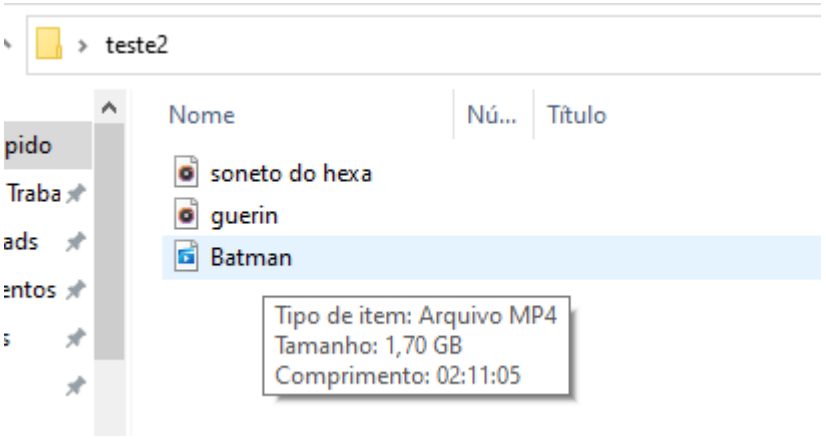
Peer 1 - o que fornecerá o arquivo



Peer 2 - o que receberá o arquivo



Arquivo baixado com sucesso!



5. Comandos para uso:

- join: 127.0.0.1, 8080, C:/…
- leave
- update: fileName
- search: fileName
- download: hostIP, hostPort, fileName

6. Conclusão

As redes peer-to-peer são muito interessantes quando se trata de transferência de arquivos, principalmente por conta de não depender de um servidor centralizado, que pode cair a qualquer momento, além de outras inúmeras vantagens, e, portanto, é uma arquitetura de rede que deve permanecer ativa por longos anos, tendo em vista que todas as aplicações de Torrent a usam.

O desenvolvimento do projeto, por sua vez, foi bem proveitoso, visto que adquiri um grande conhecimento de Threads, protocolos TCP e UDP, comunicação por Sockets, entre outros conceitos. Acredito ser um projeto de complexidade relativamente alta, que consumiu vários dias, e aproximadamente mil linhas de código para ser implementado. Além disso, a programação com threads e sockets possuem alguns detalhes que podem comprometer todo o funcionamento do sistema, sendo difícil, ainda, de se encontrar o erro. No método keepAlive, por exemplo, tive um entrave por conta da forma que estruturei o projeto para ser exibido no console, e, com isso, foi difícil adaptá-lo de forma a inserir este método, mesmo assim, fiz questão de tentar fazê-lo e até incluí-lo no relatório em questão.

Por ser um projeto de alta complexidade, creio que com mais tempo para desenvolver seria ideal, pois seria possível corrigir bugs que passam despercebidos e que poderiam ser resolvidos com uma análise mais minuciosa.

7. Referências

[1] https://gist.github.com/CarlEkerot/2693246

[2] https://docs.oracle.com/javase/7/docs/api/java/io/

[3] https://www.oficinadanet.com.br/post/14046-o-que-e-p2p-e-como-ela-funciona

[4] https://www.voitto.com.br/blog/artigo/o-que-e-rede-p2p

[5] https://stackoverflow.com/questions/2965747/why-do-i-get-an-unsupported operation exception-when-trying-to-remove-an-element-f

[6] https://www.youtube.com/watch?v=nysfXweTI7o

[7] Aulas do professor Vladimir Emiliano Rocha.

[8] https://stackoverflow.com/questions/5207162/define-a-fixed-size-list-in-java

[9] https://stackoverflow.com/questions/1844688/how-to-read-all-files-in-a-folder-from-java

[10] https://stackoverflow.com/questions/25709593/send-large-files-over-socket-java

[11] https://stackoverflow.com/questions/9395207/how-to-include-jar-files-with-java-file-and-compile-in-command-prompt