Sistemas Distribuídos
Projeto de Programação – Napster

Aluno: Lucas Eduardo De Mieri
Ra:11201723007
2Q2022

Resumo

Modelos P2P centralizados podem atuar como uma forma eficiente de viabilizar o compartilhamento entre múltiplos clientes simultaneamente, para o Napster, por exemplo, o compartilhamento é realizado unicamente pelos Peers e é remediado por um servidor central que valida a requisição e faz o controle dos Peers, sem a necessidade e ter cada um dos arquivos instalado de forma centralizada. Pode-se empregar Threads para fazer operações assíncronas em um servidor Napster, permitindo que múltiplas conexões sejam realizadas sem colisões e com pouco uso de recurso computacional, o que viabiliza a transferência de arquivos grandes entre Peers sem necessitar de um grande poder computacional de um servidor central.

1.Demonstração.

O link segue com um vídeo exemplificando o uso do Napster,em uma máquina local, onde é feita uma transferência entre Peers de diferentes sockets.
https://www.youtube.com/watch?v=aQC4lKxjtFw

2. Descrição de Alto Nível do Napster.

O Napster é um serviço que permite o compartilhamento de arquivos na modalidade centralizada do *Peer to Peer* (*P2P*), fazendo uso de uma estrutura cliente servidor, onde o servidor tem o papel de direcionar os clientes. O servidor é responsável por registrar os arquivos armazenados em cada uma das máquinas e os registros dos usuários que possuem esses arquivos, permitindo que os clientes façam requisições de busca e download para o servidor principal, que por sua vez atuará solicitando o download e disponibilizando o endereço onde o arquivo pode ser encontrado.

3. Descrição de Alto Nível do Servidor.

O Napster opera baseado em um servidor central, seu principal objetivo é armazenas o endereço de todos os usuários e intermediar a conexão entre o Peer que detém o arquivo e o Peer interessado.

3.1 Descrição de Alto Nível do NapsterServer

O NapsterServer é o servidor centralizado que recepciona todos os Peers, dentro dessa classe o método main realiza a recepção dos novos Peers, por meio da:

```
86              //Recepcão de Novos Peers, inicia um novo Peer que e reconhecido pelo servidor
87              clientReception peer = new clientReception(recPacket, serverSocket);
88              peer.start();
```

Uma vez que um novo Peer é reconhecido pelo servidor é possível que ocorra o compartilhamento de arquivos entre as pastas especificadas, agora cabe ao servidor verificar se os Peers seguem disponíveis.

```
76          // verificar a disponibilidade do peer
77          checkAlive check = new checkAlive(serverSocket);
78          check.start();
```

Sendo assim o NapsterServer, recepciona os novos Peers, principalmente, por meio dos métodos checkAlive e clientReception.

3.2 Descrição de Alto Nível do checkAlive.

O método checkAlive envia mensagens atualizando o NapsterServer sobre o status dos Peers, o método procura o nome e o socket dos peers vivos e retorna para o NapsterServer.

```
299                 try { //Peers vivos
300                     if (NapsterServer.peerAlive.get(peer)) {
301
302                         DatagramSocket SvSocket = this.serverSocket;
303                         InetAddress GetName = InetAddress.getByName(peer.split(":")[0]);
304                         int PeerInt = Integer.parseInt(peer.split(":")[1]);
305                         //envia a mensagem de alive
306                         NapsterServer.send(alive,SvSocket ,GetName ,PeerInt);
307                         //replace de status antigo
308                         NapsterServer.peerAlive.replace(peer, false);
309                         // peers mortos
310                     } else {
311                         NapsterServer.peerAlive.remove(peer);
312                         System.out.println("Peer " + peer + " morto. Eliminando seus arquivos");
313                         //array com a lista de files do peer
314                         ArrayList<String> FilestoRemove = NapsterServer.peerFiles.get(peer);
315                         // mata as informacoes do peer
316                         NapsterServer.printFiles(FilestoRemove);
317                         NapsterServer.peerAlive.remove(peer);
318                         NapsterServer.peerFiles.remove(peer);
```

Caso um Peer não esteja disponível durante a execução do checkAlive, o método irá informar ao server sobre o desligamento do Peer e por fim remove-lo da lista de peerAlive e peerFiles, excluindo informações sobre o Peer indisponível.

3.3 Descrição de Alto Nível do clientReception.

O método clientReception é o intermediário entre as requisições P2P, sendo responsável pelos status: ALIVE_OK, JOIN, SEARCH, SERACH_OK, LEAVE, LEAVE_OK, UPDATE, UPDATE_OK

```
170         if (req.equals("ALIVE_OK")) {
171             if (NapsterServer.peerAlive.containsKey(ip_porta)) {
172                try { NapsterServer.peerAlive.replace(ip_porta, true);}
173                catch (Exception e) { System.out.println(e.getMessage());
174                }
175             }
176         } else if (req.equals("JOIN")) {
177             if (!NapsterServer.peerFiles.containsKey(ip_porta)) {
178 
179                 ArrayList<String> reqToList = this.requisicao.getList();
180                 try {
181                 NapsterServer.peerFiles.put(ip_porta, reqToList);
182                 NapsterServer.peerAlive.put(ip_porta, true);}catch (Exce
183             }
184 
185 
186             resposta.setmsg("JOIN_OK");
```

Tomando como exemplo o ALIVE_OK, pode-se observar que o método valida cada uma das requisições dos usuários, afim de garantir que elas sejam devidas antes de encaminha-las para os demais Peers. O status ALIVE_OK é responsável por prontamente validar se um Peer esta disponível, varrendo a lista de ip_porta e procurando pelo ip do Peer. Já o método Join verifica a requisição de join, habilitando novos peers ao NapsterServer.

```
211         else if (req.equals("SEARCH")) {
212 
213             ArrayList<String> peerList = new ArrayList<String>();
214 
215             if (NapsterServer.peerFiles.containsKey(ip_porta)) {
216 
217                 System.out.println("Peer " + ip_porta + " solicitou ar
218 
219 
220                 Set<String> peers = NapsterServer.peerFiles.keySet();
221                 int count=0;
222                 for (String peer : peers) {
223                     if (!peer.equals(ip_porta) && (NapsterServer.peerF
224                         System.out.println("Peer numero:"+count+" cont
225                         peerList.add(peer);
226                     }
227                     count++;
228                 }
229             } else {
230                 peerList = new ArrayList<String>();
231             }
232 
233 
234             resposta.setmsg("SEARCH_OK");
235             resposta.setList(peerList);
236             DatagramSocket svSocket = this.serverSocket;
237             InetAddress requAdd = this.requisicao.getAddress();
238             int rePort = this.requisicao.getPort();
239 
240             NapsterServer.send(resposta, svSocket, requAdd,rePort );
```

Os status SEARCH e SEARCH_OK, são referentes a busca de um arquivo através dos Peers conhecidos pelo servidor, o clientReception mapeia as ip_porta e compara os arquivos disponíveis com os solicitados pelo cliente, uma vez que o arquivo é encontrado é emitido o status SEARCH_OK e o usuário é informado.

```
242         else if (req.equals("LEAVE")) {
243             if (NapsterServer.peerFiles.containsKey(ip_porta)) {
244 
245                 System.out.println("O Peer"+ip_porta+ "deixara o
246                 try {
247                 NapsterServer.peerFiles.remove(ip_porta);
248                 NapsterServer.peerAlive.remove(ip_porta);}
249                  catch (Exception e) { System.out.println(e.getMe
250                  }
251             }
252 
253 
254             resposta.setmsg("LEAVE_OK");
255             DatagramSocket sendSocket = this.serverSocket;
256             InetAddress ReqAdd = this.requisicao.getAddress();
257             int ReqPort = this.requisicao.getPort();
258             NapsterServer.send(resposta, sendSocket, ReqAdd, ReqP
259         }
260 
261         else if (req.equals("UPDATE")) {
262             if (NapsterServer.peerFiles.containsKey(ip_porta)) {
263 
264                 if (!NapsterServer.peerFiles.get(ip_porta).contai
265 
266                     ArrayList<String> updatedList = NapsterServer
267                     updatedList.add(requisicao.getList().get(0));
268                     NapsterServer.peerFiles.replace(ip_porta, upd
269                 }
270             }
271 
272 
273             resposta.setmsg("UPDATE_OK");
274             DatagramSocket sendSocket = this.serverSocket;
275             InetAddress ReqAdd = this.requisicao.getAddress();
276             int ReqPort = this.requisicao.getPort();
277             NapsterServer.send(resposta, sendSocket, ReqAdd, ReqP
278         }
279     }
```

Para os status de LEAVE e LEAVE_OK, primeiro é encontrado a key referente ao peer no NapsterServer e então seus files e status de alive é removido, uma vez que as remoções são bem-sucedidas é emitido o statuso de LEAVE_OK e a mensagem é enviada. Já para o statua de UPDATE e UPDATE_OK, a requisição é procurada na lista de files, uma vez que o peer correspondente ao file é identificado

pode-se emitir o status de UPDATE_OK e dar continuidade a requisição.

4. Descrição de Alto Nível do NapsterPeer.

O NapsterPeer é responsável por fazer o intermédio entre as requisições de usuário e o servidor, uma vez que as requisições são validadas o Peer pode se conectar a outros peers e realizar o download de arquivos.
Para o NapsterPeer são implementados os métodos: Comunication, AliveHandler, ServerHandler, peerAcessRequest, peerReception.

```
87
88              System.out.println("-----------------------------------
89              System.out.println("Menu de Opcoes");
90              System.out.println("Pressione 0 para JOIN");
91
92              System.out.println("Pressione 1 para SEARCH");
93
94              System.out.println("Pressione 2 para DOWNLOAD");
95
96              System.out.println("Pressione 3 para LEAVE");
97
98              int menu = in.nextInt();
99
100             Boolean received = false;
101
102             ArrayList<String> ListaReqs = new ArrayList<String>();
103
104             Comunication requisicao = new Comunication();
105
106
107
108                 if (menu == 0) {
109                     requisicao.setmsg("JOIN");
110                     requisicao.setList(fileNames);
111
112                     Comunication resposta = waitResponse("JOIN_OK",
113                             serverPort); // blocking
```

O NapsterPeer faz o intermédio entre as demais implementações e o usuário, recebendo suas requisições.

4.1 Descrição de Alto Nível do Comunication.

Comunicação Serializada entre Peers e servidor, contem implementações que busca as mensagens, portas e endereços.

```
423     public String getmsg() {
424         return this.msg;
425     }
426
427     public void setmsg(String m) {
428         this.msg = m;
429     }
```

4.3 Descrição de Alto Nível do AliveHandler.

O AliveHandler informa que o status de Alive do Peer para o servidor sempre que é perguntado.

```
477             resposta.setmsg("ALIVE_OK");
478             DatagramSocket msgClientSocket = this.clientSocket;
479             InetAddress msgIpAdress = this.IPAddress;
480             int msgServerPort = this.serverPort;
481             NapsterPeer.send(resposta, msgClientSocket, msgIpAdress, msgServerPort);
482             //envia a mensagem de vivo para o servidor
483         } catch (Exception e) { System.out.println(e.getMessage());
```

4.4 Descrição de Alto Nível do ServerHandler.

O ServerHandler realizar a comunicação persistente com o servidor, monitora o status de AliveHandler e notifica em caso de perda de conexão, além de atuar enviando e recebendo mensagens do servidor.

```
504     public void run() {
505         try {
506             while (true) {
507
508                 byte[] recBuffer = new byte[1024];
509                 int buffer_size=recBuffer.length;
510                 DatagramPacket recPacket = new DatagramPacket(recBuffer, buffer_size);
511                 clientSocket.receive(recPacket); // blocking
512
513                 ByteArrayInputStream byteStream = new ByteArrayInputStream(recBuffer);
514                 ObjectInputStream objectIn = new ObjectInputStream(new BufferedInputStream(byteStream
515                 Comunication resposta = (Comunication) objectIn.readObject();
516
517                 if (resposta.getmsg().equals("ALIVE")) {
518                     try {
519                     AliveHandler answer = new AliveHandler(this.clientSocket, recPacket.getAddress(),
```

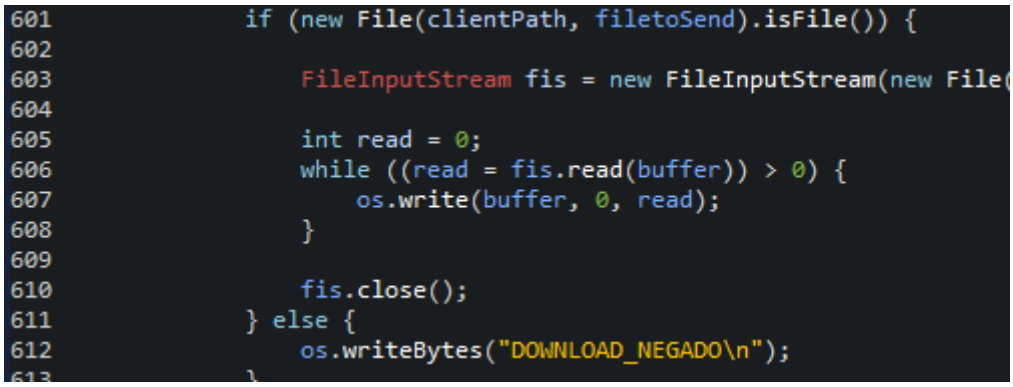
4.5 Descrição de Alto Nível do peerAcessRequest

Tenta estabelecer a comunicação entre o cliente e o servidor em caso de um novo Peer tentar se conectar ao servidor Napster, monitora a condição de aceite da requisição, persistindo até receber o retorno do servidor.

```
550         this.clientPath = filePath;
551     }
552
553     public void run() {
554
555     try {
556         ServerSocket serverSocket = new ServerSocket(this.port);
557
558
559         System.out.println("Making peer request");
560             while (true) {
561                 try {
562
563                     Socket no = serverSocket.accept();
564                     String receiveClientPath = this.clientPath;
```

4.6 Descrição de Alto Nível do peerReception

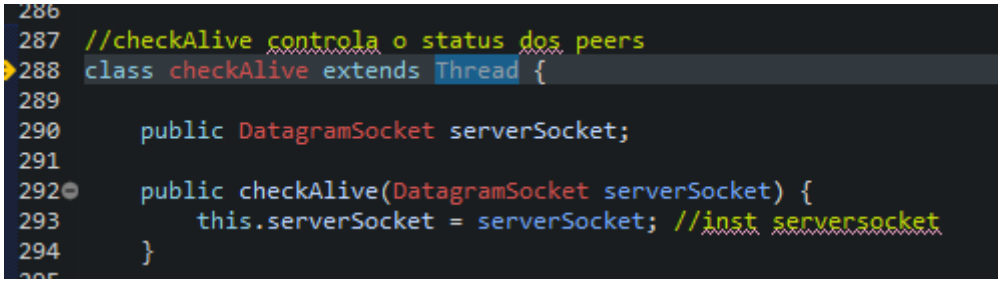
Operação P2P, recepciona um outro Peer que foi previamente aprovado pelo servidor, atuando como implementação para o Peer doador ser capaz de disponibilizar os seus arquivos para o Peer receptor.



```
601             if (new File(clientPath, filetoSend).isFile()) {
602
603                 FileInputStream fis = new FileInputStream(new File(
604
605                 int read = 0;
606                 while ((read = fis.read(buffer)) > 0) {
607                     os.write(buffer, 0, read);
608                 }
609
610                 fis.close();
611             } else {
612                 os.writeBytes("DOWNLOAD_NEGADO\n");
```

5.1 Uso de Threads para o Servidor.

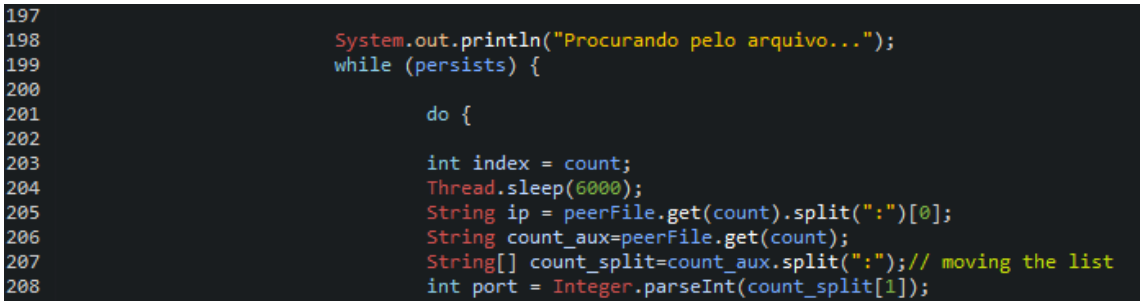
Para o servidor Threads podem ser usadas para verificar o status de alive de cada um dos Peers, as podem ser feitas de forma assíncrona, permitindo a um mesmo servidor atender diversas requisições diversas de Peers distintos.



```
286
287 //checkAlive controla o status dos peers
288 class checkAlive extends Thread {
289
290     public DatagramSocket serverSocket;
291
292     public checkAlive(DatagramSocket serverSocket) {
293         this.serverSocket = serverSocket; //inst serversocket
294     }
```

5.2 Uso de Threads para o Cliente.

Threads são utilizadas em diversos momentos para os Peers, uma das aplicações diz respeito a operações assíncronas sendo realizadas entre Peers e Servidor.
Tomando como por exemplo a busca de arquivo, dada pelo status SEARCH, quando um arquivo não é encontrado a Thread persiste em sua busca, se movendo pela lista de arquivos conhecidos até encontrar o arquivo ou atingir algum critério de parada.



```
197
198                     System.out.println("Procurando pelo arquivo...");
199                     while (persists) {
200
201                             do {
202
203                             int index = count;
204                             Thread.sleep(6000);
205                             String ip = peerFile.get(count).split(":")[0];
206                             String count_aux=peerFile.get(count);
207                             String[] count_split=count_aux.split(":");// moving the list
208                             int port = Integer.parseInt(count_split[1]);
```

6. Transferência de Arquivos Grandes.

O envio de grandes arquivos é feito por conta de um envio fracionado, ao invés de enviar um grande arquivo de uma única vez são disponibilizados “blocos” sequenciais que constituem esse arquivo, sua montagem é feita pelo Peer que recebeu o arquivo e

evita grande parte do processo de download seja perdida caso alguns Peers fique indisponível.

```
587
588    public void run() {
589
590        try {
591
592            InputStreamReader is = new InputStreamReader(this.no.getInputStream());
593            BufferedReader reader = new BufferedReader(is);
594
595            String filetoSend = reader.readLine();
596
597            DataOutputStream os = new DataOutputStream(this.no.getOutputStream());
598            byte[] buffer = new byte[1024];
599
600
601            if (new File(clientPath, filetoSend).isFile()) {
602
603                FileInputStream fis = new FileInputStream(new File(clientPath, filetoSend).getAbsolutePath());
604
605                int read = 0;
606                while ((read = fis.read(buffer)) > 0) {
607                    os.write(buffer, 0, read);
```

7. Referências

- Vídeo explicativo sobre programação UDP, TCP e Threads: https://www.youtube.com/watch?v=nysfXweTI7o

- Informações sobre programação com UDP podem ser encontradas em:
  https://www.baeldung.com/udp-in-java
  https://www.geeksforgeeks.org/working-udp-datagramsockets-java/
  https://docs.oracle.com/javase/tutorial/networking/datagrams/index.html

- Informações sobre programação com TCP podem ser encontradas em:
  https://www.baeldung.com/a-guide-to-java-sockets

- Informações sobre Threads, que permitem que o servidor ou peer receba e envie informações de forma simultânea:
  https://www.baeldung.com/a-guide-to-java-sockets (Seção 6: TCP com muitos clientes)
  https://www.tutorialspoint.com/java/java_multithreading.htm

- Informações sobre o funcionamento do Napster:
  https://www.britannica.com/topic/Napster

https://arxiv.org/ftp/cs/papers/0402/0402018.pdf
https://computer.howstuffworks.com/napster.htm