UNIVERSIDADE FEDERAL DO ABC
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

RELATÓRIO EP1 - SISTEMAS DISTRIBUÍDOS

FELIPE OLIVEIRA SILVA
RA: 11201822479

Profª. Drª. Rodrigo Izidoro Tinini

1 Video

É possível conferir o funcionamento do código em 4 peers através do link: https://rebrand.ly/urm90tm

2 Funcionamento

O objetivo dessa prática foi criar um sistema distribuido P2P não-estruturado que procura um arquivo em um dos peers conhecidos. Cada peer é inicializado contendo endereço IP e porta própria e de outros dois *peers*, a comunicação "corre" através desses peers até chegar no destinátario que possui o arquivo. Caso encontre, é enviada uma mensagem diretamente para o *peer* que originou a busca. Esse protocolo é conhecido como "Gossip".

Foram implementados os mecanismos de envio e recebimento de mensagens UDP, leitura de arquivos e timeout. O código-fonte utiliza 4 threads (*watchdog*, *sendMessage*, *handler* e *timeout*) para paralelizar as ações. Cada uma dessas threads tem um papel especifico.

2.1 Menu interativo

O primeiro passo do código é criar o "menu interativo" onde é possível de fazer a inicialização do *peer* e salvar os dados dos outros 2 peers conhecidos digitando o número 1. Em seguida, digitando o número 2, é feita a busca do arquivo nos peers conhecidos. Para para captar a tecla digitada, é utilizada a função *Scanner.nextInt().*

```
public static void main(String args[]) throws Exception {
    // menu
    Scanner in = new Scanner(System.in);
    DatagramSocket serverSocket = null;

    printMenu();

    while (true) {
        try {
            int option = in.nextInt();

            if (option == 1) { // INICIALIZA…
            } else if (option == 2) { // SEARCH…
            } else {
                System.out.println(x: "Opção invalida!");
            }
        } catch (Exception e) {
            System.out.println(x: "Opção invalida!");
        }
    }
```

2.2 Função *INICIALIZA*

Em um processo bem parecido com o menu interativo, é feita a coleta dos dados de IP e porta do *peer* e dos outros 2 peers conhecidos. Nesse caso, usando a função *Scanner.nextLine()* para o IP e *path* e o *Scanner.nextInt()* para a porta.

```
Scanner input = new Scanner(System.in);

// peer address
System.out.print(s: "Entre com o endereco IP deste peer (ex: 127.0.0.1): ");
String peerIp = input.nextLine();
if (peerIp == "") {
    peerIp = "127.0.0.1";
}
peerAddress = InetAddress.getByName(peerIp);
```

Depois da coleta, é impresso os arquivos contidos no *path* do *peer* através da função *printArray(fileNames).* Por fim, são iniciadas 2 threads separadas, a classe *PeerWatchdog,* para imprimir a cada 30 segundos os arquivos contidos na pasta e a classe *PeerHandler* que lida com o recebimento e encaminhamento das mensagens. A linha *serverSocket = new DatagramSocket(peerPort)* é responsável por inicializar a conexão socket para recebimento das mensagens e é atribuida na variável da classe *Peer*. A classe *PeerHandler* recebe essa conexão como parâmetro.

```
// print files in the folder each 30s
PeerWatchdog wd = new PeerWatchdog(peerIp, peerPort, folderPath);
wd.start();

// start a socket connection in the typed peer port
serverSocket = new DatagramSocket(peerPort);

// listen messages and forward
PeerHandler peer = new PeerHandler(serverSocket, folderPath);
peer.start();
```

A classe *PeerHandler* é responsável por receber todas as mensagens UDPs através da função *this.serverSocket.receive(recPacket).* Em seguida o BufferStream do pacote UDP é convertida em objeto do tipo *Mensagem* e atribuido à variável *UDPRequest.* Para definir corretamente o endereço IP e porta da requisição, é chamada a função *UDPRequest.addAddress* que insere em uma lista e define as variáveis *port* e *address* da classe *Mensagem*.

```java
static class PeerHandler extends Thread {
    public Mensagem UDPRequest;
    public DatagramSocket serverSocket;
    private String folderPath;
    private ArrayList<String> filesProceeded = new ArrayList<>();
    private ArrayList<String> ipPortProceeded = new ArrayList<>();
    private ArrayList<String> dateTimeProceeded = new ArrayList<>();

    public PeerHandler(DatagramSocket serverSocket, String folderPath) {
        this.serverSocket = serverSocket;
        this.folderPath = folderPath;
    }

    public void run() {

        // waits peers UDP contact
        while (true) {
            try {
                byte[] recBuffer = new byte[1024];
                DatagramPacket recPacket = new DatagramPacket(recBuffer, recBuffer.length);
                this.serverSocket.receive(recPacket);

                // deserialize Mensagem object
                ByteArrayInputStream byteStream = new ByteArrayInputStream(recPacket.getData());
                ObjectInputStream objectIn = new ObjectInputStream(new BufferedInputStream(byteStream));
                Mensagem UDPRequest = (Mensagem) objectIn.readObject();

                // get IP and port from peer
                UDPRequest.addAddress(recPacket.getAddress(), recPacket.getPort());
```

Em seguida é iniciada uma nova *Mensagem,* que será a resposta para essa requisição recebida. Então, é mantido em uma lista todos os endereços IPs e portas que já transitaram a mensagem com o mesmo IP e porta de origem e o mesmo *datetime.*

```java
// compose a new forward message
Mensagem UDPResponse = new Mensagem();

// copy address list from request to response
UDPResponse.setAddressList(this.UDPRequest.getAddressList(), this.UDPRequest.getPortList());

// keep same origin in response
UDPResponse.setOrigin(this.UDPRequest.getOriginAddress(), this.UDPRequest.getOriginPort());

// keep same origin message date time
UDPResponse.setDateTime(dateTime);
```

Se a mensagem for do tipo "SEARCH", é verificado se a mensagem não está sendo processada novamente para o mesmo IP, porta e *datetime* de origem e se ela já não foi encaminhada para um dos dois peers conhecidos. Se isso ocorrer, é lançado um erro com um print "requisição já processada" e não há uma mensagem de resposta.

```java
if (message.equals(anObject: "SEARCH")) { // SEARCH
    Boolean fileFound = false;
    File[] listFiles = new File(this.folderPath).listFiles();
    String ipPort = "";

    String peer1IpPort = getIpPort(Peer.peerAddress1, Peer.peerPort1);
    String peer2IpPort = getIpPort(Peer.peerAddress2, Peer.peerPort2);

    boolean peer1IsProceeded = this.isProceeded(fileName, peer1IpPort, dateTime.toString());
    boolean peer2IsProceeded = this.isProceeded(fileName, peer2IpPort, dateTime.toString());
    boolean originPeerProceeded = this.isProceeded(fileName, this.UDPRequest.getOriginIpPort(),
            dateTime.toString());

    // check if origin peer message was proceeded and next known peer too
    if (originPeerProceeded && peer1IsProceeded && peer2IsProceeded) {
        System.out.println("requisição já processada para " + fileName);
        throw new Exception();
    }
```

Caso a requisição não tenha sido processada, são listados e lidos em um laço de repetição os arquivos daquele *peer* e verificado se o nome do arquivo existe. Se ele existir, é enviado diretamente a mensagem do tipo "RESPONSE" para o *peer* que requisitou aquele arquivo (endereço de IP e porta de origem) por meio da função *Peer.sendMessage*.

```java
for (File f : listFiles) {
    if (f.isFile() && fileName.equals(f.getName())) {
        // send message to origin peer
        InetAddress address = this.UDPRequest.getOriginAddress();
        int port = this.UDPRequest.getOriginPort();
        ipPort = this.UDPRequest.getOriginIpPort();

        UDPResponse.setMessage(m: "RESPONSE");
        UDPResponse.setFileName(fileName);
        UDPResponse.addAddress(address, port);

        fileFound = true;

        System.out.println("tenho " + fileName + " respondendo para " + ipPort);
        Peer.sendMessage(UDPResponse, serverSocket, address, port);
        break;
    }
}
```

Se o arquivo não se encontra naquele *peer,* é sorteado aleatoriamente um novo peer (entre os dois conhecidos) para reencaminhar a mensagem. Caso um dos dois *peer,* já tenha sido processado, automaticamente é escolhido o *peer* em que a mensagem ainda não tenha sido processada. A mensagem é encaminhada com o tipo "SEARCH" e inserida em 3 listas de requisição processadas (nome de arquivos, IPs/portas, *datetime*).

```java
if (!fileFound) {
    // randomly get a peer to retry request
    int peerIndex = Peer.getRandomPeerIndex();
    InetAddress peerA;
    int peerP;
    if (peerIndex == 0 && peer1IsProceeded) {
        peerIndex = 1;
    } else if (peerIndex == 1 && peer2IsProceeded) {
        peerIndex = 0;
    }

    if (peerIndex == 0) {
        peerA = Peer.peerAddress1;
        peerP = Peer.peerPort1;
    } else if (peerIndex == 1) {
        peerA = Peer.peerAddress2;
        peerP = Peer.peerPort2;
    } else {
        throw new Exception();
    }

    UDPResponse.setMessage(m: "SEARCH");
    UDPResponse.setFileName(fileName);

    // keep origin message datetime
    UDPResponse.addAddress(peerA, peerP);
    ipPort = UDPResponse.getIpPort();

    // add to proceeded list (to avoid resend same message)
    this.filesProceeded.add(fileName);
    this.ipPortProceeded.add(ipPort);
    this.dateTimeProceeded.add(dateTime.toString());

    System.out.println("não tenho " + fileName + " respondendo para " + ipPort);
    Peer.sendMessage(UDPResponse, serverSocket, peerA, peerP);
}
```

Para finalizar, no caso das requisições de *Mensagem* do tipo 'RESPONSE", é impresso na tela uma mensagem que o arquivo foi encontrado e é retirado da fila de *timeout.*

```java
} else if (message.equals(anObject: "RESPONSE")) {
    System.out
            .println("peer com arquivo procurado: " + this.UDPRequest.getIpPort() + " " + fileName);
    Peer.timeoutProceedQueue.remove(dateTime.toString());
}
```

2.3 Função *SEARCH*

O objetivo da função *SEARCH* (opção 2 do menu interativo) é buscar e descobrir em qual *peer* está um determinado arquivo. Para isso, novamente com a função *input.nextLine()* é lido qual nome do arquivo para realizar a busca. Em seguida é criada uma nova instância da classe *Mensagem* com o objetivo de escrever as informações da mensagem à ser enviada, tais como o tipo da mensagem (*SEARCH* ou *RESPONSE*), nome do arquivo à ser buscado, IP e porta de origem da mensagem. Logo é inserido o *datetime* da mensagem em uma fila de *timeout.* Por fim, é escolhido um *peer* aleatório (entre os dois inseridos na função *INICIALIZA)* e é criada duas threads, a primeira dentro da função *sendMessage* (responsável por fazer o envio da mensagem UDP) e a outra thread responsável por checar se a mensagem excedeu o tempo limite de *timeout* (10 segundos).

```
Scanner input = new Scanner(System.in);

// filename
System.out.print(s: "Informe o nome do arquivo para buscar (ex: video.mp4): ");
String fileName = input.nextLine();

// send request and name of file to search
Mensagem UDPRequest = new Mensagem();
UDPRequest.setMessage(m: "SEARCH");
UDPRequest.setFileName(fileName);
UDPRequest.setOrigin(peerAddress, peerPort);

// add origin datetime to timeout queue
timeoutProceedQueue.add(UDPRequest.getDateTime().toString());

int peerIndex = getRandomPeerIndex();
InetAddress peerA;
int peerP;
if (peerIndex == 0) {
    peerA = peerAddress1;
    peerP = peerPort1;
} else {
    peerA = peerAddress2;
    peerP = peerPort2;
}

// send UDP message to random known peer
sendMessage(UDPRequest, serverSocket, peerA, peerP);

// start timeout of message
waitResponse(serverSocket, fileName, UDPRequest.getDateTime().toString());
```

```java
public static void sendMessage(Mensagem req, DatagramSocket socket, InetAddress ip, int port) {
    new Thread(() -> {
        try {
            ByteArrayOutputStream byteStream = new ByteArrayOutputStream(size: 1024);
            ObjectOutputStream objectOut = new ObjectOutputStream(new BufferedOutputStream(byteStream));
            objectOut.flush();
            objectOut.writeObject(req);
            objectOut.flush();
            byte[] sendData = byteStream.toByteArray();

            DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, ip, port);
            socket.send(sendPacket);
            objectOut.close();
        } catch (Exception e) {
            System.out.println(e);
        }
    }).start();
}

private static void waitResponse(DatagramSocket s, String f, String dt) {
    int timeout = 10000; // time in miliseconds

    // periodically check response timeout
    new Thread(() -> {
        int t = 0;
        boolean timeoutFinished = false;

        while (!timeoutFinished) {
            try {
                Thread.sleep(millis: 100);
                t = t + 100;

                // check if the timeout is reached and the message isn't get a response
                if (t >= timeout && isInTimeoutQueue(dt)) {
                    timeoutFinished = true;
                    timeoutProceedQueue.remove(dt);
                    System.out.println("ninguém no sistema possui o arquivo " + f);
                }

            } catch (Exception e) {

            }
        }
    }).start();
}
```