DIEGO SOUSA SANTOS – 11044616

UFABC

RELATÓRIO EP-GOSSIP – SISTEMAS DISTRIBUÍDOS

SÃO CAETANO DO SUL – SP

10/2022

LINK DO VÍDEO: https://youtu.be/hI4q6p4p0vo

1. FORMATO DA MENSAGEM TRANSFERIDA

As mensagens são transferidas através do objeto Mensagem, descrito no diagrama UML abaixo:

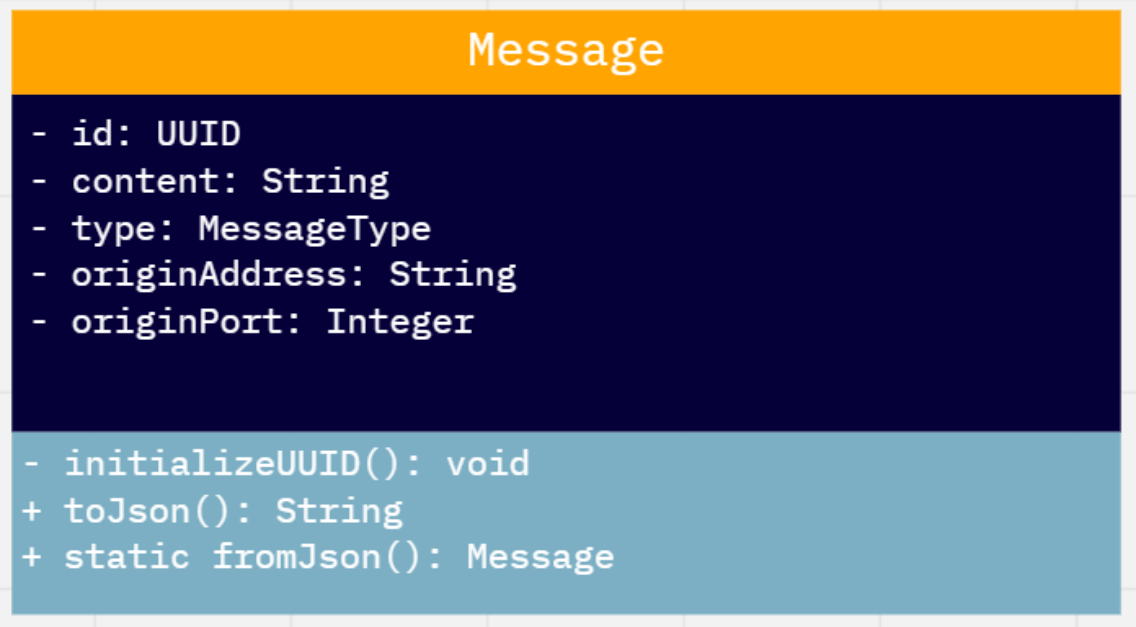

Figura 1 UML da classe Message

O atributo “id” define o identificador da mensagem através de um UUID que permite tornar única cada mensagem.

O atributo “contente” armazena o conteúdo da mensagem. No nosso caso, o arquivo que está sendo procurado.

O atributo “type” é um struct que classifica a mensagem como SEARCH ou RESPONSE, possibilitando diferenciar esses dois tipos de mensagens.

O atributo “originAddress” armazena o endereço IP do Peer de origem da mensagem.

O atributo “originPort” armazena a porta do Peer de origem da mensagem.

O método “initializeUUID” gera o UUID da mensagem.

O método “toJson” utiliza a biblioteca gson para converter o objeto Mensagem em Json.

O método estático “fromJson” retorna uma Mensagem a partir de um json.

2. PEERS E SUAS FUNCIONALIDADES

2.1 MÉTODO MAIN

O método main é definido logo no início da classe Peer. Nele existe um loop para leitura dos dados do usuário. É impresso um menu composto das opções “1. INICIALIZA” e “2. SEARCH”, as quais o usuário pode selecionar informando o número ou o texto da opção.

Caso a opção de inicializar seja selecionada, o usuário deverá informar o endereço IP, a porta, o diretório de arquivos e duas portas de peers vizinhos para o novo peer que será criado. Com isso, uma nova instância de um objeto peer é criada e manipulado nas consultas de search.

Caso a opção de search seja selecionada, o usuário deverá informar o nome do arquivo que será buscado na função de search.

```
// INICIALIZA ------------
if (option == "INICIALIZA") {

    if(peer != null) {
        peer.closeSocket();
        peer = null;
    }

    // ler ip
    String address = "";
    while (address == null || address.isEmpty()) {
        System.out.print("Informe o IP: ");
        address = keyboard.next();
    }

    // ler porta
    String port = "";
    while (port == null || port.isEmpty()) {
        System.out.print("Informe a Porta: ");
        port = keyboard.next();
    }

    // ler a pasta dos arquivos
    String filesFolder = "";
    System.out.print("Informe o diretório dos arquivos: ");
    while (filesFolder == null || filesFolder.isEmpty() || Files.notExists(Paths.get(filesFolder))) {
        filesFolder = keyboard.nextLine();
    }

    // ler porta de peer vizinho
    String port1 = "";
    while (port1 == null || port1.isEmpty()) {
        System.out.print("Informe a porta de outro peer (1): ");
        port1 = keyboard.next();
    }

    // ler porta de peer vizinho
    String port2 = "";
    while (port2 == null || port2.isEmpty()) {
        System.out.print("Informe a porta de outro peer (2): ");
        port2 = keyboard.next();
    }

    // inicializa peer
    peer = new Peer(Integer.parseInt(port), address, filesFolder);

    // cria o socket
    try {
        peer.createSocket();
    } catch (Exception e) {
        System.out.println("Falha ao inicializar Socket do Peer " + address + " | " + port);
        throw e;
    }

    // lista arquivos do diretório
    peer.setFilesPaths();
    // adiciona os vizinhos
    peer.addNeighbors(Arrays.asList(new Integer[] { Integer.parseInt(port1), Integer.parseInt(port2) }));
}
```

Figura 2 código para inicializar peer

```java
// SEARCH -------------------------
else if (option == "SEARCH") {
    // ler nome do arquivo buscado
    String fileName = "";
    while (fileName == null || fileName.isEmpty()) {
        System.out.print("Informe o nome do arquivo desejado: ");
        fileName = keyboard.nextLine();
    }

    // criando e enviando mensagem de busca
    Message searchMessage = new Message(fileName, MessageType.SEARCH);
    searchMessage.setOriginAddress(peer.getAddress());
    searchMessage.setOriginPort(peer.getPort());

    byte[] sendData = new byte[1024];
    sendData = searchMessage.toJson().getBytes();
    peer.forwardMessageToNeighbor(searchMessage);
}
```

Figura 3 código para buscar arquivos em peers

2.2 DEFINIÇÃO DO PEER

Os peers são definidos através da classe descrita no diagrama UML abaixo:

Figura 4 diagrama UML do Peer

2.2.1 ATRIBUTOS

O atributo “port” define a porta em que o socket do peer é criado, enquanto o atributo “address” define o endereço do socket.

O atributo “filesFolder” armazena o diretório de arquivos do peer, enquanto o atributo lista “filesPaths” armazena os nomes dos arquivos desse diretório.

O atributo “neighbors” registra as portas dos peers vizinhos ao peer em questão.

Os objetos “timerFiles” e “timerTimeOut” são cronômetros assíncronos (via thread” para realizar as rotinas de atualizar a lista de arquivos e calcular timeout, respectivamente.

A lista “receivedMessagesControl” faz a gestão das mensagens recebidas para evitar duplicidade de tratamento. Já o Map (ou dicionário) “sentMessagensControl” faz a gestão das mensagens enviadas, armazenando o horário do envio para posterior tratamento de timeout.

O objeto “serverSocket” é o DatagramSocket utilizado pelos peers para se comunicar, e a constante TIMEOUT_SECONDS define o limite máximo de tempo em segundos para determinar o timeout.

2.2.2 MÉTODOS

O método closeSocket() apenas realiza o fechamento do socket de um peer.

```
// destrói o peer e as conexões/threads por ele criadas
public void closeSocket() {
    serverSocket.close();
    serverSocket = null;
    timerFiles.cancel();
    timerTimeOut.cancel();
}
```

Figura 5 método closeSocket()

O método createSocket(), por sua vez, inicializa o serverSocket e cria um loop assíncrono (linhas 211 à 278) para receber as diversas requisições que podem chegar simultaneamente. Quando uma mensagem é recebida em formato de json, ela é convertida em um objeto mensagem e é realizada uma verificação na lista receivedMessagesControl para garantir que a mensagem já não está sendo tratada pelo peer. Caso contrário ela é descartada.

Depois disso, caso a mensagem seja do tipo SEARCH, é verificado se no diretório do peer existe o arquivo informado no contente da mensagem. Se o arquivo existe, é enviada a resposta ao peer de origem através do método answerToOriginPeer. Caso contrário, a mensagem é redirecionada através do método forwardToNeighbor.

Caso a mensagem seja do tipo RESPONSE, o peer exibe na tela que o arquivo foi encontrado e a mensagem é removida do controle de mensagens enviadas.

```java
// cria o socket que vai recepcionar as mensagens
public void createSocket() throws Exception {
    Thread th = new Thread(() -> {
        // canal de comunicação não orientado à conexão
        serverSocket = null;
        try {
            serverSocket = new DatagramSocket(port);
        } catch (SocketException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        //System.out.println("Peer inicializado.");

        while (true && serverSocket != null) {
            // buffer de recebimento
            byte[] recBuffer = new byte[1024];
            // datagrama que será recebido
            DatagramPacket recPacket = new DatagramPacket(recBuffer, recBuffer.length);

            try {
                // recebendo pacote
                serverSocket.receive(recPacket);
                String informacao = new String(recPacket.getData(), recPacket.getOffset(), recPacket.getLength());
                // mensagem recebida no pacote
                Message message = Message.fromJson(informacao);
                // verifica se a mensagem já não foi tratada
                if (receivedMessagesControl.stream().anyMatch(m -> m.getId().equals(message.getId())))
                    continue;

                receivedMessagesControl.add(message);

                // criando mensagem de resposta
                Message responseMessage = new Message();

                // Se é uma mensagem de busca...
                if (message.getType() == MessageType.SEARCH) {
                    // verifica se o arquivo existe no diretório
                    Boolean fileExists = fileExistsInFolder(message.getContent());
                    // se existe, envia para o peer de origem
                    if (fileExists) {
                        answerToOriginPeer(message);
                        // remove a mensagem da lista de tratadas depois de um tempo
                        Timer tempTimer = new Timer();
                        tempTimer.schedule(new TimerTask() {
                            @Override
                            public void run() {
                                receivedMessagesControl.remove(message);
                                tempTimer.cancel();
                            }}, 0, 1000);
                    } else {
                        // se não existe, passa para os vizinhos
                        forwardMessageToNeighbor(message);
                    }
                }
                // se é uma mensagem de resposta, apenas printa e faz gestão da fila de mensagens enviadas
                else {
                    String responseText = String.format("Peer com o arquivo procurado: %s:%s %s",
                            recPacket.getAddress(), recPacket.getPort(), message.getContent());
                    System.out.println(responseText);
                    sentMessagesControl.remove(message);
                }

            } catch (IOException e) {
                if(!e.getMessage().toUpperCase().equals("Socket Closed".toUpperCase()))
                    e.printStackTrace();
            }
        }
    });

    th.start();
}
```

Figura 6 método createSocket()

O método answerToOriginPeer() é responsável por criar uma nova mensagem de RESPONSE utilizando o endereço e a porta de origem da mensagem de SEARCH para sinalizar ao peer de origem que o arquivo foi encontrado.

```java
    // método para responder ao peer de origin da solicitação
    private void answerToOriginPeer(Message message) throws IOException {
        Message responseMessage = new Message();
        String responseText = String.format("Tenho %s. Respondendo para %s:%s", message.getContent(),
                message.getOriginAddress(), message.getOriginPort());
        System.out.println(responseText);
        // formatando mensagem de resposta
        responseMessage.setContent(message.getContent());
        responseMessage.setType(MessageType.RESPONSE);
        responseMessage.setOriginAddress(message.getOriginAddress());
        responseMessage.setOriginPort(message.getOriginPort());
        responseMessage.setId(message.getId());

        // buffer de resposta
        byte[] sendBuffer = new byte[1024];
        sendBuffer = responseMessage.toJson().getBytes();
        // pacote da resposta
        DatagramPacket sendPacket = new DatagramPacket(sendBuffer, sendBuffer.length,
                InetAddress.getByName(message.getOriginAddress()), message.getOriginPort());
        serverSocket.send(sendPacket);
    }
```

Figura 7 método answerToOriginPeer

O método forwardMessageToNeighbor() é responsável por criar uma thread (linha 306) que selecionará aleatoriamente os peers vizinhos (linhas 314 à 316) para encaminhar a mensagem de SEARCH para eles, desde que não sejam o peer de origem da mensagem. A partir dessa seleção, é criado o datagrampacket para transportar a mensagem e ela é retransmitida ao próximo peer e adicionada ao controle de mensagens enviadas (linhas 334 à 346).

```java
    // método GOSSIP de proparação da mensagem, selecionando aleatoriamente o próximo peer comunicado
    public void forwardMessageToNeighbor(Message message) {

        Thread th_forward = new Thread(() -> {
            // lista para mapear peers não comunicados
            List<Peer> checkedPeers = new ArrayList<Peer>(neighbors);

            // varrendo vizinhos
            while (checkedPeers.size() > 0) {

                // pegando peer randômico e adicionando na lista de visitados
                Random rand = new Random();
                Peer randNeighbor = checkedPeers.get(rand.nextInt(checkedPeers.size()));
                checkedPeers.remove(randNeighbor);

                // não redireciona se o vizinho aleatório é o peer de origem
                if(randNeighbor.getPort().equals(message.getOriginPort()))
                    continue;

                InetAddress IPAddress = null;
                // criando datagram socket
                try {
                    IPAddress = InetAddress.getByName(randNeighbor.getAddress());
                } catch (UnknownHostException e1) {
                    // TODO Auto-generated catch block
                    e1.printStackTrace();
                }

                // criando packet
                byte[] sendData = new byte[1024];
                sendData = message.toJson().getBytes();
                DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, IPAddress, randNeighbor.port);
                // enviando packet
                try {

                    if(!this.port.equals(message.getOriginPort())) {
                        String responseText = String.format("não tenho %s. Encaminhando para %s:%s", message.getContent(),
                                InetAddress.getByName(randNeighbor.getAddress()), randNeighbor.getPort());
                        System.out.println(responseText);
                    }

                    serverSocket.send(sendPacket);
                    // inserindo mensagem no controle de mensagens já tratadas
                    sentMessagesControl.put(message, LocalDateTime.now());

                } catch (IOException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }

        });

        th_forward.start();
    }
```

Figura 8 método forwardMessageToNeighbor

O método fileExistsInFolder() verifica se o arquivo solicitado na mensagem existe no diretório do peer. Para isso, esse método concatena os nomes dos arquivos em uma string, separados por espaço, e verifica se o nome do arquivo procurado está nessa string.

```
358
359     private Boolean fileExistsInFolder(String file) {
360         file = new String(file);
361         return String.join(" ", filesPaths).contains(file);
362     }
363
```

Figura 9 método fileExistsInFolder

O método addNeighbors adiciona os peers vizinhos informados pelo usuário à lista de vizinhos do peer em questão.

```
417     public void addNeighbors(List<Integer> ports) {
418         for (Integer port : ports) {
419             Peer peer = new Peer(port);
420             this.neighbors.add(peer);
421         }
422     }
423     // endregion
```

Figura 10 método addNeighbors

Os temporizadores citados anteriormente são inicializados no construtor do Peer. O timerFiles faz o scheduler de uma função que lê os arquivos do diretório do peer, preenche a lista filesPaths e exibe na tela a mensagem “Sou peer x:y com os arquivos z”.

O timerTimeOut faz o scheduler de uma função que a cada segundo visita o map sentMessagesControl para procurar mensagens que foram enviadas a TIMEOUT_SECONDS segundos ou mais, calculando isso a partir da data de envio da mensagem. Caso encontre, retira a mensagem dos controles e alerta sobre o timeout.

```
160         // Scheduler de tarefa de atualização de arquivos
161         timerFiles = new Timer();
162         timerFiles.schedule(new TimerTask() {
163             @Override
164             public void run() {
165                 String paths = "";
166                 setFilesPaths();
167 
168                 if (filesPaths != null && filesPaths.size() > 0)
169                     paths = String.join(" | ", filesPaths);
170 
171                 System.out.println(String.format("Sou peer %s:%s com arquivos %s", address, port, paths));
172             }
173         }, 0, 1000 * 30);
174 
175         // Scheduler para verificação de timeout a cada 1 segundo
176         timerTimeOut = new Timer();
177         timerTimeOut.schedule(new TimerTask() {
178             @Override
179             public void run() {
180                 // percorre o MAP de controle de mensagens enviadas
181                 for(Message message: sentMessagesControl.keySet()) {
182                     // calcula há quanto tempo a mensagem foi enviada
183                     long dateDiff = sentMessagesControl.get(message).until(LocalDateTime.now(), ChronoUnit.SECONDS);
184                     // se superou o tempo do timeout, imprime a mensagem, remove do controle de envio e insere no controle de
185                     if(dateDiff >= TIMEOUT_SECONDS) {
186                         System.out.println("Ninguém no sistema possui o arquivo " + message.getContent());
187                         sentMessagesControl.remove(message);
188                         receivedMessagesControl.add(message);
189                     }
190                 }
191             }
192         }, 0, 1000);
193 
```

Figura 11 definição dos timers