U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Faculdade de Engenharia da Universidade do Porto

Sistemas Distribuídos

3º ANO - **MIEIC**

Distributed **Backup** Service for the **Internet**

Estudantes & Autores

Ana Mafalda Santos, 201706791

201706791@fe.up.pt

David Freitas Dinis, 201706766

201706766@fe.up.pt

Diogo Silva, up201706892

up201706892@fe.up.pt

João Henrique Luz, up201703782

up201703782@fe.up.pt

22 de **maio** de **2020**

sdis1920-t2g21

Visão Geral (*Overview*)

No âmbito da unidade curricular de Sistemas Distribuídos foi-nos proposta a realização de um projeto de ***Distributed Backup Service for the Internet*** utilizando ***Java***. Este relatório tem por objetivo documentar o *design* e a arquitetura implementada para tornar possíveis as funcionalidades de ***backup***, ***delete***, ***restore*** e ***reclaim***, assim como esclarecer quais as medidas tomadas para garantir a escalabilidade e tolerância a falhas.

Para este projeto optámos por utilizar:

- **JSSE** que permite uma comunicação segura.
- ***Thread Pools*** para tornar possível a concorrência do programa.
- ***Java NIO*** para a leitura e escrita em ficheiro.
- **Arquitetura *Chord*** para garantir a escalabilidade do programa.

Protocolos

Para a comunicação entre o *client* e o *peer*, foi usado o protocolo **RMI**. Entre *peers,* todas as comunicações são feitas através do protocolo **TCP**, com JSSE através do uso de **SSLSokets**.

Subprotocolo Backup

Neste protocolo, o *peer* ao receber a mensagem BACKUP[1] pelo *client*, codifica o nome do ficheiro, obtendo só o seu identificador (*fileId*) e guardando-o, de seguida, numa *synchronizedList* de forma a manter um registo de todos os *backups* por si iniciados.

De acordo com o seu identificador (*peerId*) e o *peerId* do seu predecessor, verifica se o *fileId* está compreendido entre estes. Caso seja verdade, envia a mensagem BACKUP ao seu sucessor.

Caso o *fileId* não esteja compreendido nos valores mencionados anteriormente, este envia a mensagem FORWARD[2] a um dos *peers* da sua ***Finger Table***, ou seja, o que tem o seu *peerId* mais semelhante ao *fileId*.

Caso um *peer* receba BACKUP, verifica se é o *home peer*, se tem espaço para o armazenar e se já tem o ficheiro guardado. No primeiro e segundo caso (não ter espaço), reencaminha o BACKUP para o seu sucessor. Na última situação, reduz em um a *replicationDegree* atual e, caso ainda seja

[1] `BACKUP <ip_address> <port> <successor_ip_address> <successor_port> <file_id> <replication_degree> <body_length> <body>`

[2] `FORWARD <file_id> <replication_degree> <body_length> <body>`

maior que zero, reencaminha a mensagem ao seu sucessor.

Após passar por estas verificações iniciais, significa que tem as condições necessária para guardar o ficheiro, e assim o faz. Após guardar o ficheiro, envia duas mensagens de confirmação – mensagens STORED[3] – usando os endereços IP (*ipAddress*) recebidos na mensagem. Esta mensagem serve para o *peer* cujo identificador é imediatamente a seguir a esse *fileId* e o seu sucessor, guardarem num ***ConcurrentHashMap<BigInteger, List<OutsidePeer>>*** quem são os *peers* que guardaram os ficheiros. Verifica ainda se é necessário reencaminhar a mensagem, retirando uma unidade à grau de replicação e, caso este ainda seja superior a zero, envia a mensagem com a esse grau atualizado para o seu sucessor. Podemos ver isto no seguinte excerto de código:

```
final Path fileDirPath = Paths.get(fileDirName + "/" + fileKey);

Files.newOutputStream(fileDirPath, StandardOpenOption.WRITE, StandardOpenOption.CREATE).write(body);
System.out.println("File size: " + body.length + "bytes");

System.out.println("Stored!");
replicationDegree--;

OutsidePeer peer = new OutsidePeer(inetSocketAddress);
this.peer.getStorage().addStoredFile(new BigInteger(fileKey));
Messenger.sendStored(new BigInteger(fileKey), myIpAddress, myPort, inetSocketAddress);
Messenger.sendStored(new BigInteger(fileKey), myIpAddress, myPort, successorInetSocketAddress);

if (replicationDegree >= 1) {
    message = "BACKUP " + ipAddress + " " + port + " " + succesorIpAddress + " " + successorPort + " "
            + fileKey + " " + replicationDegree + " " + body.length + "\n";
    this.peer.sendMessage(message, body, outsidePeer.getInetSocketAddress());
}
```

Um *peer* pode ainda receber a mensagem FORWARD, caso em que vai verificar se o seu *fileId* está compreendido entre o seu *peerId* e o do seu predecessor. Se a condição não se verificar, reencaminha a mensagem FORWARD, inalterada, para o seu sucessor. Caso se verifique, o *peer* envia a mensagem BACKUP para os seus sucessores. A mensagem FORWARD serve, então, para encontrar o *peer* que vai guardar num *ConcurrentHashMap* a lista de *peers* que guardou o ficheiro: *peer* cujo *peerId* é imediatamente a seguir ao *fileId*.

Subprotocolo de *Restore*

Para realizar o protocolo *restore* foram criadas três novas mensagens: a mensagem

[3] `STORED <file_id> <ip_address> <port>`

RESTORE[4], FINDFILE[5] e GIVEFILE[6].

Quando um *peer* recebe o pedido *restore* do *client*, procede à codificação do nome do ficheiro, criando o fileId e enviando a mensagem RESTORE a um dos *peers* da sua Finger Table (o que tem o seu peerId mais semelhante ao fileId).

Sempre que a mensagem restore é recebida, o *peer* verifica se tem o ficheiro armazenado. Caso o tenha, envia a mensagem GIVEFILE ao *peer* que iniciou o pedido (usando o endereço IP e porta da mensagem RESTORE recebida). Se não tiver o ficheiro, reencaminha a mensagem para o seu sucessor.

Caso a mensagem RESTORE seja recebida pelo *peer* cujo *peerId* é imediatamente após o *fileId*, este, usando a informação guardada no *ConcurrentHashMap*, envia a mensagem FINDFILE aos *peers* que guardaram o ficheiro. Esta mensagem é mandada a um de cada vez. No caso de a resposta deste não ser positiva (SENT) ou ser inexistente, este envia a mensagem a outro dos *peers*. Isto pode ser visto na função seguinte:

```
public boolean getFile(BigInteger fileId, String ipAddress, int port) throws IOException {
    List<OutsidePeer> peers = fileLocation.get(fileId);
    String message = new String();
    String response = new String();
    SSLSocket sslSocket = null;

    for (int i = 0; i < peers.size(); i++) {
        InetSocketAddress socket = peers.get(i).getInetSocketAddress();
        // FINDFILE file_key ip_address port
        message = "FINDFILE " + fileId + " " + ipAddress + " " + port + "\n";
        sslSocket = Messenger.sendMessage(message, socket);
        BufferedReader in = new BufferedReader(new InputStreamReader(sslSocket.getInputStream()));
        if (sslSocket.isInputShutdown()) {
            continue;
        }
        response = in.readLine();
        in.close();
        if (response.equals("SENT " + fileId)) {
            return true;
        }
    }

    return false;
}
```

Ao receber a mensagem FINDFILE, o *peer* verifica se tem o ficheiro. Se o tiver, responde ao

[4] `RESTORE <file_id> <ip_address> <port>`
[5] `GIVEFILE <fileKey> <body_length> <body>`
[6] `FINDFILE <file_id> <initiator_peer_ip_address> <initiator_peer_port>`

pedido com SENT e envia a mensagem GIVEFILE ao *peer* iniciador.

Quando o *peer* iniciador recebe a mensagem GIVEFILE, este guarda a informação recebida num ficheiro, terminando assim o protocole de restauro.

Subprotocolo de *Deletion*

Neste protocolo, o *initiator peer* recebe o pedido DELETE enviado pelo *cliente* e verifica se tem o ficheiro armazenado. Caso o tenha, elimina-o. Depois, verifica também se o identificador do ficheiro está contido entre o seu próprio ID e o ID do seu predecessor. Em caso negativo, envia a mensagem DELETE[7] para outro *peer*. Caso a condição se verifique, acede ao *ConcurrentHashMap fileLocation* onde tem a informação dos *peers* que fizeram *backup* do ficheiro e envia a cada um deles a mensagem DELETE. Podemos ver de seguida a função encarregada desta ação:

```
public boolean sendDelete(BigInteger fileId) throws IOException {
    List<OutsidePeer> peers = fileLocation.get(fileId);
    String message = new String();
    if(peers != null){
        for (int i = 0; i < peers.size(); i++) {
            InetSocketAddress socket = peers.get(i).getInetSocketAddress();
            // FINDFILE file_key ip_address port
            message = "DELETE " + fileId + "\n";
            Messenger.sendMessage(message, socket);
        }
    }

    return true;
}
```

Sempre que um *peer* recebe esta mensagem, verifica se tem o ficheiro guardado e se o contém nalguma das *synchronizedList* (*storedFiles* e *askedFiles*) usadas para manter a informação do *peer*. Se algum destes tiver o *fileId*, este mesmo registo será eliminado dessas estruturas.

[7] `DELETE <file_id> <ip_address> <port> | DELETE <file_id>`

Subprotocolo de *Reclaim*

Neste protocolo, o *initiator peer* ao receber o pedido de *reclaim* do cliente, começa por atualizar o valor do espaço disponível para armazenamento. De seguida, verifica se necessita de apagar ficheiros alguns dos ficheiros que armazena. Se for o caso, começa por apagar ficheiros um a um até deixar de ter espaço ocupado superior ao disponível, sendo que os ficheiros são apagados de forma aleatória. Para cada ficheiro apagado é enviada a mensagem REMOVED[8] usando a *Finger Table* disponível. Esta mensagem é enviada para o *peer* com identificados mais semelhante ao *fileId* do ficheiro apagado.

Sempre que um *peer* recebe a mensagem REMOVED, este verifica se é o *peer* com *peerId* imediatamente a seguir ao *fileId* enviado e, se não o for, reencaminha a mensagem. Se o for, apaga este *peer* do *ConcurrentHashMap*, enviando também ao seu sucessor a mensagem REMOVELOCATION[9] para que este atualize, do mesmo modo, o seu *HashMap*. Por fim, envia BACKUP para o seu sucessor. Podemos encontrar o código que executa estas operações abaixo:

```
// Peer that holds table with file locations
if (Helper.middlePeer(fileKey, peer.getPredecessor().getId(), peer.getId())) {

    String message1 = "REMOVELOCATION " + request[1] + " " + request[2] + " " + request[3] + "\n";

    String message = "BACKUP " + this.peer.getAddress().getAddress().getHostAddress() + " "
            + this.peer.getAddress().getPort() + " "
            + this.peer.getSuccessor().getInetSocketAddress().getAddress().getHostAddress() + " "
            + this.peer.getSuccessor().getInetSocketAddress().getPort() + " " + fileKey + " " + "-1 "
            + body.length + "\n";
    try {
        if (!outsidePeer.testSuccessor()) {
            this.peer.sendMessage(message, body, outsidePeer.getInetSocketAddress());
            this.peer.sendMessage(message1, body, outsidePeer.getInetSocketAddress());
        } else {
            OutsidePeer otherSuccessor = this.peer.getNextSuccessor();
            this.peer.sendMessage(message, body, otherSuccessor.getInetSocketAddress());
            this.peer.sendMessage(message1, body, otherSuccessor.getInetSocketAddress());
        }
    } catch (IOException e) {
    }
    return "OK\n";
```

[8] REMOVED <file_id> <ip_address> <port> <body_length> <body>
[9] REMOVELOCATION <file_id> <ip_address> <port>

Design de Concorrência

O *design* de concorrência implementado suporta a execução simultânea de várias instâncias dos diferentes subprotocolos. Tal deve-se não só à forma como são recebidas e tratadas as mensagens enviadas pelos *peers* envolvidos, mas também às restrições impostas à escrita e leitura de informação por parte dos *threads* lançados. Usamos então *Thread Pools* na comunicação, sendo algumas lançadas após a sua criação, enquanto que outras são agendadas para executarem instruções periódicas.

```
ipAddress = address;
this.id = Helper.getPeerId(ipAddress, port); // chord.hashSocketAddress(address);
this.port = port;
this.listener = new RequestListener(this);
executor = (ScheduledThreadPoolExecutor) Executors.newScheduledThreadPool(150);
executor.execute(this.listener);
this.stabilizer = new Stabilizer(this);
executor.scheduleAtFixedRate(stabilizer, 5, 5, TimeUnit.SECONDS);
```

Para lidar com ficheiros, utilizamos *Java NIO* que, entre outras vantagens, impede que um ficheiro bloqueie ao ser acedido por mais do que uma *thread*.

```
try {
    final Path filePathDir = Paths.get(folderDirectory);
    final Path filePath = Paths.get(fileDirectory);
    if (Files.notExists(filePathDir))
        Files.createDirectories(filePathDir);

    Files.newOutputStream(filePath, StandardOpenOption.WRITE, StandardOpenOption.CREATE).write(body);

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

JSSE

Todas as mensagens trocadas entre *peers* utilizam JSSE (Java Secure Socket Extension). Isto quer dizer que, todas as mensagens descritas anteriormente são enviadas utilizando esta forma de comunicação pela Internet. Assim, garantimos que todas as comunicações estão a ocorrer de forma segura, protegendo-se não só as identificações dos *peers,* mas também o conteúdo das

mensagens enviadas. Para isto, foram utilizadas as principais classes de JSSE *SSLServerSocket* e *SSLSocket* do *package javax.net.ssl*.

Do lado do cliente:

```
public class Messenger {
    public static SSLSocket sendMessage(String message, InetSocketAddress socket) {
        SSLSocketFactory sslSocketFactory = (SSLSocketFactory) SSLSocketFactory.getDefault();
        SSLSocket sslSocket = null;
        try {
            sslSocket = (SSLSocket) sslSocketFactory.createSocket(socket.getAddress().getHostAddress(),
                    socket.getPort());

            DataOutputStream out = new DataOutputStream(sslSocket.getOutputStream());
            out.writeBytes(message);
            out.flush();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return sslSocket;
    }
```

Como podemos ver, para criar um *SSLSocket* é necessário instanciar uma *SSLSocketFactory* e criar um *SSLSocket* com a *port* e o *ipAddress* pretendido. Por último é instanciado um *DataOutputStream* com o objetivo de enviar as mensagens ao servidor.

Do lado do servidor, classe *RequestListener* e *RequestHandler*, respetivamente:

```
@Override
public void run() {
    final SSLServerSocketFactory sslServerSocketFactory = (SSLServerSocketFactory) SSLServerSocketFactory
            .getDefault();
    SSLServerSocket sslServerSocket = null;

    try {
        sslServerSocket = (SSLServerSocket) sslServerSocketFactory.createServerSocket(peer.getPort());
    } catch (final IOException e) {
        e.printStackTrace();
    }

    while (true) {
        SSLSocket sslSocket;
        // System.out.println("listining");

        try {
            sslSocket = null;
            sslSocket = (SSLSocket) sslServerSocket.accept();
            peer.getExecutor().execute(new RequestHandler(peer, sslSocket));
        } catch (final IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
    @Override
    public void run() {
        try {
            DataOutputStream out = new DataOutputStream(sslSocket.getOutputStream());
            BufferedReader in = null;

            String responseMess = null;
            in = new BufferedReader(new InputStreamReader(sslSocket.getInputStream()));

            responseMess = in.readLine();

            String[] request = responseMess.split(" ");

            String response = "\n";
            byte[] file;

            switch (request[0]) {
```

Neste caso, é instanciado uma *SSLServerFactory* e também um *SSLServerSocket*. Para ler o pedido vindo do cliente é usado um *BufferedReader*.

Escalabilidade

De forma a garantir a máxima escalabilidade do nosso projeto, procuramos fazer um bom uso das várias ferramentas ao nosso dispor. Assim, optamos por implementar a arquitetura *Chord*, que nos permite gerir informação numa rede escalável computadores; utilizar *Thread Pools;* e a leitura e escrita assíncronas de ficheiros utilizando a *Java NIO*, tal como descrito na secção do ***Design* de Concorrência**.

Passamos agora a uma breve análise dos esforços feitos na nossa implementação da arquitetura *Chord*. Como mencionamos, este *design* permite a gestão de dados presentes em múltiplos computadores (cada um deles representado por um nó ou, vulgarmente, um *peer*) que interagem numa rede. Esta arquitetura garante ainda a integridade da informação frente a possíveis variações bruscas do número de nós envolvidos (assunto abordado na próxima secção). Uma vez que através deste método (baseado em *Hash Tables*) é criado um mapeamento entre cada nó e uma chave correspondente, este algoritmo permite que a pesquisa de um determinado nó na rede seja feita com complexidade temporal logarítmica *O(log(N))*, tornando-o numa ótima estratégia para conseguir um sistema escalável.

Para que seja feito o mapeamento dos identificadores (chaves) aos nós presentes, é aplicada a função criptográfica SHA-1. Além disto, é possível garantir que a probabilidade de colisão na atribuição destas chaves é desprezável com o uso de *Consistent Hashing*. Este mapeamento é feito tanto para os *peers* como para os ficheiros que lhes serão atribuídos consoante a chave correspondente.

Uma vez devidamente identificados, os *peers* podem integrar o *Chord Ring* (anel/círculo formado pelos vários nós ativos). Para formar este círculo, é necessário que um primeiro *peer* fique ativo e à escuta de mensagens. Cada novo *peer* terá então de contactar um *peer* que já tenha integrado o círculo para descobrir a sua posição dentro deste. Esta posição é atribuída em função da chave calculada na primeira fase. Uma vez contactado um *peer* ativo no círculo, este inicia uma pesquisa pelo nó que será sucessor do *peer* que o contactou. Esta pesquisa é feita recorrendo ao método *findSuccessor*:

```
private void findSuccessor(String[] request) throws UnknownHostException, IOException {
    OutsidePeer newPeer = new OutsidePeer(new InetSocketAddress(request[2], Integer.parseInt(request[3])));
    // FINDSUCCESSOR <peer_key> <ip_address> <port>
    // Second peer to join
    if (this.peer.getSuccessor() == null) {
        this.peer.setSuccessor(newPeer);
        this.peer.setPredecessor(newPeer);
        this.peer.getFingerTable().add(this.peer.getSuccessor(), 0);

        Messenger.sendUpdatePosition(this.peer.getAddress().getAddress().getHostAddress(), this.peer.getPort(),
                this.peer.getAddress().getAddress().getHostAddress(), this.peer.getPort(),
                newPeer.getInetSocketAddress());
    }
    // New peer is between him and his successor
    else if (Helper.middlePeer(new BigInteger(request[1]), this.peer.getId(), this.peer.getSuccessor().getId())) {
        Messenger.sendUpdatePosition(this.peer.getAddress().getAddress().getHostAddress(), this.peer.getPort(),
                this.peer.getSuccessor().getInetSocketAddress().getAddress().getHostAddress(),
                this.peer.getSuccessor().getInetSocketAddress().getPort(), newPeer.getInetSocketAddress());
        this.peer.setSuccessor(new OutsidePeer(new InetSocketAddress(request[2], Integer.parseInt(request[3]))));
    }
    // The position of the new peer isn't known
    else {
        Messenger.sendFindSuccessor(new BigInteger(request[1]), request[2], Integer.parseInt(request[3]),
                this.peer.getSuccessor().getInetSocketAddress());
    }
}
```

Este método inicia uma cadeia de mensagens (do tipo FINDSUCCESSOR) entre os *peers*, até que seja encontrado o *peer* que será o sucessor.

Cada um dos *peers* implementados regista informação relevante de forma a saber com que *peers* deve comunicar, nomeadamente o seu sucessor, o seu antecessor e a *Finger Table*.

A *Finger Table* trata-se de uma estrutura de dados na qual ficam registadas as informações necessárias para contactar até um máximo de *m* nós no *Chord Ring*. Nesta implementação recorremos a objetos da classe *OutsidePeer,* que armazenam e gerem dados como o endereço de IP, a porta e a chave do *peer* que representam. Operações elementares como adição, remoção, atualização e pesquisa nesta estrutura são feitas recorrendo à classe *FingerTable*. É, portanto, graças a esta tabela que é possível a pesquisa eficiente de nós.

Periodicamente, e sem que isso interfira com outras operações que estejam a decorrer, é também necessário levar a cabo o processo de estabilização. Neste processo, são feitas as correções necessárias para que o sistema se mantenha robusto face à entrada e saída de novos *peers*. Estas correções integram as medidas tomadas para a tolerância a falhas, como explicado mais à frente.

Esta operação é realizada por objetos da classe *Stabilizer*, que implementa a interface *Runnable* e que executam as suas funções de forma concorrente.

```
public void run() {
    try {
        System.out.println("Peer with id: " + this.peer.getId());
        System.out.println("Successor id: " + this.peer.getSuccessor().getId());
        System.out.println("Predecessor id: " + this.peer.getPredecessor().getId());
        this.peer.getFingerTable().print();
        peer.getStorage().print();
        if (peer.getSuccessor() != null) {
            if (peer.getSuccessor().testSuccessor()) {
                if (peer.updateToNextPeer()) {
                    return;
                }
            }
            OutsidePeer newNextPeer = this.peer.getSuccessor().getNextSuccessor();
            if (newNextPeer.getId().compareTo(this.peer.getId()) == 0) {
                this.peer.setNextSuccessor(null);
            } else {
                this.peer.setNextSuccessor(newNextPeer);
            }
            peer.updateTable();
            peer.getSuccessor().notifySuccessor(peer.getAddress(), peer.getSuccessor().getInetSocketAddress());
            peer.stabilize();
            peer.getExecutor().execute(fingerFixer);
        }
    } catch (Exception e) {

    }
}
```

Tolerância a Falhas

Na nossa implementação tomamos medidas para evitar pontos de falha no caso algum dos *peers* efetuar uma saída inesperada do sistema. Para tal, usamos variadas técnicas:

- **Utilização de um determinado *replication degree:***

  Começamos por assegurar que no *backup* dos ficheiros é usado sempre um valor igual ou, idealmente, superior a dois neste parâmetro. Desta forma, é possível garantir que mesmo que um *peer* que estivesse a guardar o ficheiro seja encerrado inesperadamente, ainda existirá pelo menos uma outra cópia desse ficheiro no sistema, garantindo também que será possível efetuar o seu restauro.

- **Armazenamento dos diferentes dados em vários *peers:***

  Várias estruturas de dados (como *ConcurrentHashMap e synchronizedList)* são usadas para armazenar informações acercas dos *peers e ficheiros já guardados.*

- **Métodos de sincronização dos dados do sistema:**

  Para manter as informações referidas no ponto anterior atualizadas, assim como assegurar a integridade e bom funcionamento do serviço, foram necessários adicionar alguns métodos de sincronização. Em primeiro lugar, cada *peer* trata de averiguar junto

do seu sucessor se o predecessor deste ainda é ele e assim perceber se é necessário atualizar tal dado. Nesse caso, são alteradas também a *Finger Table*, o seu sucessor, o seu predecessor, entre outros mencionados mais à frente. Para além disso, para evitar a ocorrência de *starvation* dos *threads*, usamos a função *setSoTimeout* do *SSLSocket* para saber se o nosso sucessor está ativo ou não. Se este estiver inativo, atualizamos o nosso sucessor para o próximo *peer*.

Com esta implementação, no protocolo de *restore,* o *peer* que era suposto ter o ficheiro guardado, pode por vezes não coincidir com o que realmente tem. Para resolver este problema, fizemos uso de uma *Finger Table* para procurar o *peer* como o identificador imediatamente a seguir ao do ficheiro, i.e., o que tem o seu *peerId* mais semelhante ao *fileId*. Se este não contiver esse ficheiro, passamos ao seu sucessor, e assim sucessivamente, até haver algum que o contenha ou até ser encontrado o *peer* que pediu inicialmente esse mesmo *restore*. Se esta última situação acontecer, é assumido que o ficheiro não se encontra de todo no sistema.

- **Registo do Sucessor do Sucessor:**

Este registo é feito, pois facilita a atualização do *peer* sucessor quando este fica inativo. Quando isto acontece, é enviada a mensagem UPDATEPREDECESSOR[10] ao sucessor do sucessor para este atualizar o seu antecessor. Para isto funcionar corretamente, temos de atualizar este sucessor do sucessor constantemente. Caso este seja nulo, quer dizer que só existem dois *peers* no sistema e se mais algum *peer* sair do sistema, o sucessor ficará também nulo, voltando assim ao estado inicial.

[10] `UPDATEPREDECESSOR <ip_predecessor> <port_predecessor>`