Faculdade de Engenharia da Universidade do Porto

Computação Paralela e Distribuída

Relatório - Segundo Projeto

Distributed and Partitioned Key-Value Store

Tomás Gonçalves up201806763@edu.fe.up.pt
Alexandra Ferreira up201806784@edu.fe.up.pt
Vasco Teixeira up201802112@edu.fe.up.pt

1 - Descrição do Problema

O segundo projeto, realizado no âmbito da unidade curricular da Computação Paralela e Distribuída, tem como objetivo o desenvolvimento de um armazenamento persistente de key-value distribuído para um grande cluster. Key-value simplesmente subentende um armazenamento onde objetos de dados arbitrários são guardados e associados a uma chave, o que posteriormente permite o seu acesso. De modo a garantir a persistência, os pares key-value devem ser guardados em armazenamento persistente, como numa HDD ou num SSD. Por distribuído, entende-se que o armazenamento é particionado entre diferentes nós do cluster. O serviço está pronto para lidar com solicitações simultâneas e tolerar diversas falhas, como falhas de nós e perda de mensagem.

Para poder testar o armazenamento key-value, foi necessário desenvolver um TestClient. Essencialmente, o que este deve permitir fazer é invocar qualquer um dos eventos de *membership* (*join* ou *leave*), bem como invocar qualquer uma das operações em pares de key-value (*put* (guarda ficheiro no sistema), *get* (lê ficheiro do sistema) e *delete* (apaga ficheiro do sistema)).

Foi usada a linguagem de programação Java.

2 - Membership Service

Ao inicializar um node enviamos-lhe um endereço e uma porta multicast, estes são usados para criar uma socket multicast numa classe à qual chamamos de “Cluster Manager”. Esta classe implementa a interface runnable. Quando chamamos o seu método “run”, esta fica à espera de receber mensagens multicast, quando recebe uma mensagem, cria uma instância da classe (também runnable) “MessageManager” e executa-a. Esta classe por sua vez tem o intuito de fazer a análise das mensagens de membership recebidas e processá-las.

```
public void run(){

    try{
        this.mCastSocket = new MulticastSocket(this.mCastPort);
        this.mCastSocket.joinGroup(this.mCastAddress);

        while(true){
            byte[] buffer = new byte[2048];
            DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
            mCastSocket.receive(packet);

            String content = node.getRidOfAnnoyingChar(packet);

            node.getExecutor().execute(new MessageManager(this.node, content));
        }

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

Quando recebe uma mensagem de join o node em questão atualiza o seu membership log(formato: node_id-membership_counter; exemplo: 1-2) e a membership list(lista com os membros que atualmente fazem parte do cluster). Depois envia um pedido de conexão ao node que primeiramente enviou a mensagem de join e sendo a conexão estabelecida envia-lhe o membership log atualizado.

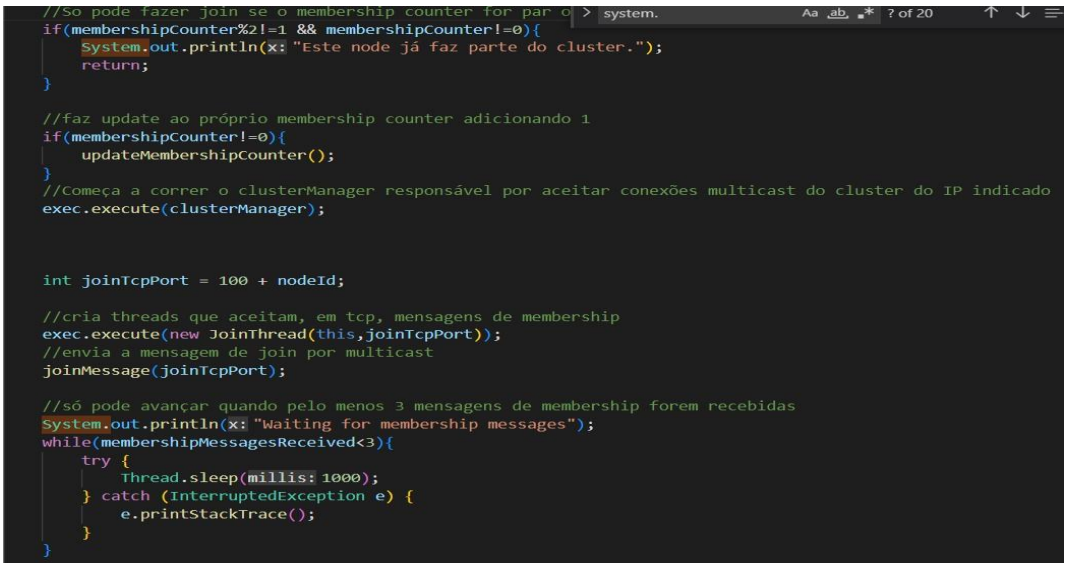
```
public void receiveJoinMessage(String node_id, String membershipCounter, int port, InetAddress tcpAddress){
    /*recebe a mensagem, retira IP e MembershipCounter,
    faz update ao membership log, e envia o membership log de volta ao node que fez join*/
    System.out.println("Received join message from node " + node_id);
    try{
        node.updateMembershipLog(node_id+"-"+membershipCounter);
        node.updateMembershipList();
        Thread.sleep(millis: 1000);
        String messageString = node.getMembershipLogString();
        Thread.sleep((long) ((Math.random()* (3000 - 1000)) + 1000));
        tcpJoinConnection(messageString, tcpAddress, port);

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

```
public void tcpJoinConnection(String message, InetAddress tcpAddress, int port){
    try{
        Socket socket = new Socket(tcpAddress, port);
        PrintWriter output = new PrintWriter(socket.getOutputStream());
        output.print(message);
        output.flush();
        output.close();
        socket.close();
    }catch(Exception ex){
        System.out.println(x: "Node is no longer accepting connections..");
        //ex.printStackTrace();
    }
}
```

Quando recebe uma mensagem de “leave” o node faz update ao seu membership log para registar a operação no log e atualizar a membership list.

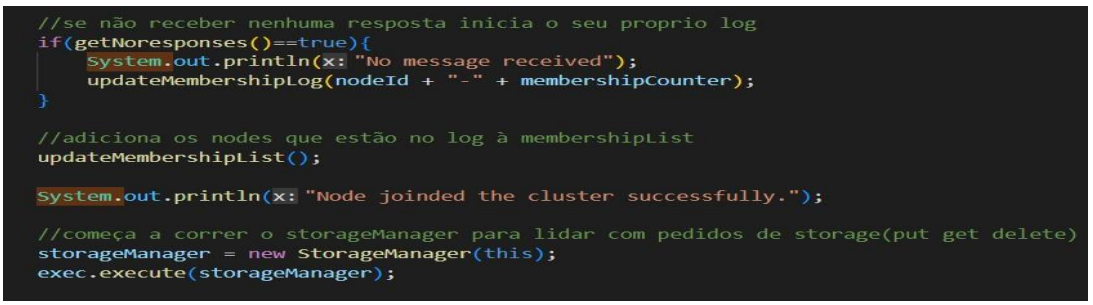
```
public void receiveLeaveMessage(String node_id, String membership_counter){
    System.out.println("Received leave message from node " + node_id + "|counter:" + membership_counter);
    node.updateMembershipLog(node_id+"-"+membership_counter);
    node.updateMembershipList();
}
```

Um node quando recebe a indicação do “client” para fazer “join” ao cluster, irá primeiro verificar se já existe um ficheiro de membership counter. Caso exista lê-o e atualiza-o. De seguida utiliza um executor para correr o cluster manager acima referido, e fica à espera de 3 conexões por tcp dos membros do cluster.



```
//So pode fazer join se o membership counter for par o
if(membershipCounter%2!=1 && membershipCounter!=0){
    System.out.println(x: "Este node já faz parte do cluster.");
    return;
}

//faz update ao próprio membership counter adicionando 1
if(membershipCounter!=0){
    updateMembershipCounter();
}
//Começa a correr o clusterManager responsável por aceitar conexões multicast do cluster do IP indicado
exec.execute(clusterManager);


int joinTcpPort = 100 + nodeId;

//cria threads que aceitam, em tcp, mensagens de membership
exec.execute(new JoinThread(this,joinTcpPort));
//envia a mensagem de join por multicast
joinMessage(joinTcpPort);

//só pode avançar quando pelo menos 3 mensagens de membership forem recebidas
System.out.println(x: "Waiting for membership messages");
while(membershipMessagesReceived<3){
    try {
        Thread.sleep(millis: 1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

Se receber atualiza o log com a informação proveniente dos nodes que receberam a sua mensagem de join. Caso não receba nenhuma conexão, após alguns segundos o node assume que não existe mais nenhum node no cluster fecha as sockets e atualiza o seu próprio log e membership list com a informação que conhece. Inicializa também o storageManager (classe runnable, responsável por enviar e receber mensagens relativas ao storage service) e utilizando o executor chama o método run da classe.



```
//se não receber nenhuma resposta inicia o seu proprio log
if(getNoresponses()==true){
    System.out.println(x: "No message received");
    updateMembershipLog(nodeId + "-" + membershipCounter);
}

//adiciona os nodes que estão no log à membershipList
updateMembershipList();

System.out.println(x: "Node joinded the cluster successfully.");

//começa a correr o storageManager para lidar com pedidos de storage(put get delete)
storageManager = new StorageManager(this);
exec.execute(storageManager);
```

Para além disto, no caso de existir mais do que um node na lista de membros o node envia um pedido ao node sucessor para um novo cálculo e reajustamento dos ficheiros guardados por esse node.

```
if(membershipList.size()>1){
    Float currentNodeAngle = getAngle(getSha256fromKey(nodeId).toString()); //angulo do novo node
    ArrayList<Float> anglesList = getAllAngles(membershipList); //lista com todos os angulos
    Collections.sort(anglesList); //ordena a lista
    Map<Float, Integer> map = getAngleIdMap(membershipList);//id atraves do angulo

    //vai buscar o node acima
    int index = anglesList.indexOf(currentNodeAngle);
    Float upper;
    if (index == anglesList.size() - 1) {
        upper = anglesList.get(index: 0);
    } else {
        upper = anglesList.get(index + 1);
    }

    int idUpper = map.get(upper);//hash do id

    System.out.println("Getting files from node " + map.get(upper));

    sendMoveMessage(String.valueOf(idUpper), currentNodeAngle);
}
```

Quando a instrução de leave é chamada pelo client, o node faz update ao membership counter e envia uma mensagem de leave para o cluster. Atualiza o seu log para conter a nova operação e faz uma transmissão dos ficheiros que tem guardados para o node sucessor que faz parte do cluster. Finalmente faz update da sua lista de membros e chama o método stop do cluster manager para que a socket multicast faça “leavegroup()” e o node sai assim oficialmente do cluster.

```
public void leave() {
    //So pode fazer leave se o membership counter for impar o que significa que já faz parte do cluster
    if(membershipCounter%2!=0){
        System.out.println(x: "Node isn't in the cluster");
        return;
    }

    updateMembershipCounter();
    writeMembershipCounter();

    leaveMessage();

    updateMembershipLog(this.nodeId + "-" + this.membershipCounter);

    if(membershipList.size()>1){
        Float currentNodeAngle = getAngle(getSha256fromKey(nodeId).toString()); //angulo do novo node
        ArrayList<Float> anglesList = getAllAngles(membershipList); //lista com todos os angulos
        Collections.sort(anglesList); //ordena a lista
        Map<Float, Integer> map = getAngleIdMap(membershipList);//id atraves do angulo

        //vai buscar o node acima
        int index = anglesList.indexOf(currentNodeAngle);
        Float upper;
        if (index == anglesList.size()-1) {
            upper = anglesList.get(index: 0);
        } else {
            upper = anglesList.get(index+1);
        }
        int idLower = map.get(upper);//hash do id

        System.out.println("Transfering files to node " + map.get(upper));
        BigInteger hashedID = getSha256fromKey(nodeId);
```

```
        File dir = new File("./" + hashedID + "/originals");
                File[] directoryListing = dir.listFiles();
                if (directoryListing != null) {
                    for (File child : directoryListing) {
                        BigInteger hashedKeyDecimal = new BigInteger(child.getName().replace(target: ".txt", replacement: ""), radix: 16);

                            try {
                                DatagramSocket sender = new DatagramSocket(new InetSocketAddress(port: 0));
                                String content = new String(Files.readAllBytes(Paths.get("./" + hashedID + "/originals/"+ child.getName())), StandardCharsets.UTF_8);
                                String value = convertStringToHex(content);
                                String original_message = "MOVEFILE" + "-" + child.getName().replace(target: ".txt", replacement: "") + "-" + value;
                                // PUT-id-key-value
                                byte[] msgBytes = original_message.getBytes();
                                InetAddress addr = InetAddress.getByName("127.0.0.1" + idLower);
                                int port = 8080;
                                InetSocketAddress dest = new InetSocketAddress(addr, port);
                                DatagramPacket hi = new DatagramPacket(msgBytes, msgBytes.length, dest);
                                sender.send(hi);
                                System.out.println("Movefile message sent to node " + idLower + "| key: " + child.getName());
                                child.delete();
                                sender.close();
                            } catch (Exception e) {
                                e.printStackTrace();
                            }

                    }
                }
    }
    updateMembershipList();
    clusterManager.stop();
    System.out.println(x: "Node left the cluster");
```

2.1 - RMI

Utilizamos o RMI para a comunicação entre o Client e o Node, no que toca ao membership service.

```
if (args.length == 2) {
    try {

        Registry registry = LocateRegistry.getRegistry();
        ClusterMembership membership = (ClusterMembership) registry.lookup(nodeAP);

        switch (operation) {
            case "join":
                membership.join();
                break;

            case "leave":
                membership.leave();
                break;
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
```

```
try {

    Node obj = new Node(mcastAddress, mcastPort, nodeId, storagePort);

    ClusterMembership stub = (ClusterMembership) UnicastRemoteObject.exportObject(obj, port: 0);

    String nodeAP = args[2];

    // Bind the remote object's stub in the registry
    Registry registry = LocateRegistry.getRegistry();
    registry.rebind(nodeAP, stub);

    System.err.println(x: "Node Ready");

} catch (Exception e) {
    System.err.println("Server exception: " + e.toString());
    e.printStackTrace();
}
```

3 - Storage Service

Antes de mais, calcula-se um mapa contendo para cada nó ativo o par (SHA-256(*nodeId*), *nodeId*). Isto permite-nos, a qualquer momento, encontrar o id de um nó através através do seu id hashed. De seguida, é igualmente gerada uma lista ordenada de cada chave SHA-256(*nodeId*). Para cada *hashed ID* de $[0;2]^{256}$, é calculado um ângulo proporcional de [0;360]; isto facilita as comparações de posições de IDs sem haver a necessidade de recorrer a cálculos com com números de 256 bits (através de variáveis BigInteger).

A operação *put* é chamada da seguinte forma:

```
TestClient <nodeId> put <"ficheiro.txt">
```

Esta função lê o conteúdo do ficheiro passado como parâmetro “ficheiro.txt”. Aquando a sua chamada, é calculada uma *key* como sendo o hash SHA-256 do conteúdo do ficheiro (convertida para decimal para facilitar o cálculo do ângulo). O *value* associado a essa *key* é o value do ficheiro convertendo o texto numa string hexadecimal; isto para poder ser feito o envio do ficheiro como sendo um só argumento, evitando problemas de formatação com os espaços no corpo do texto.

De seguida é enviada uma mensagem *"PUT-nodeId-key-value-bool"* para o nó com identificador igual a *nodeId*. O valor booleano no final da mensagem é verdadeiro quando é efetuado o PUT original e false para as réplicas. Isto evita que os PUTs das réplicas façam também réplicas delas próprias, causando um loop infinito.

O *parse* da mensagem neste caso começa por calcular o ângulo da *key hashed* do ficheiro em questão. Encontra na lista ordenada o ângulo do nó que fica imediatamente acima da posição da *key* e utiliza o mapa para encontrar o ID original do nó. Após este ter sido recuperado, assume esse nó como responsável pelo ficheiro a guardar. Se for o próprio nó em questão responsável por armazenar o ficheiro, então este trata de reconverter o hexadecimal recebido para recuperar o texto original e guarda-o no diretório “(SHA-256(*nodeId*))/originals/(*value*).txt”. Senão, o nó reenvia a mensagem PUT para o nó responsável.

A operação *get* é chamada da seguinte forma:

```
TestClient <nodeId> get <key>
```

Esta função recupera o conteúdo do ficheiro cuja chave seja *key* e que já tenha sido armazenado no sistema. É enviada uma mensagem *"GET-key"* para o nó com identificador igual a *nodeId*.

O *parse* da mensagem neste caso começa por ler a *key* do ficheiro. Subsequentemente, calcula o nó responsável. Caso seja outro nó, o próprio nó verifica na sua pasta “*/replica*” se tem uma réplica do ficheiro, senão, reenvia a mensagem GET para o responsável.

Por fim, ao receber esta, o responsável envia uma nova mensagem para o nó que

inicialmente fez o pedido GET com o formato "*RESPONSEGET-value*" sendo value o conteúdo do ficheiro em bytes. Desta forma garantimos que o conteúdo do ficheiro é sempre recebido e apresentado pelo nó inicial (responsável pelo primeiro pedido). Sempre que um nó envia um pedido RESPONSEGET, faz também print do conteúdo do ficheiro de forma a que o utilizador possa observar o que está a ser enviado pelo detentor do ficheiro.

A operação *delete* é chamada da seguinte forma:

```
TestClient <nodeId> delete <key>
```

Esta função remove o conteúdo do ficheiro cuja chave seja *key* e que já tenha sido armazenado no sistema. É enviada uma mensagem *"DELETE-key-true"* para o nó com identificador igual a *nodeId*. Este nó que recebe a mensagem, trata de a reenviar para todos os nós do sistema. Este segundo parâmetro, à semelhança do que acontece no PUT, é verdadeiro para a primeira mensagem de delete e falso para as mensagens seguintes. Desta forma apenas o que recebe o parâmetro true a reenvia para os outros, evitando loops infinitos. Cada nó, ao receber a mensagem calcula o responsável. Se for ele próprio, remove o ficheiro da sua pasta "*/originals*", senão procura na sua pasta "/*replica*" e remove-o caso exista.

4 - Replication

De modo a aumentar a disponibilidade aquando das chamadas *get*, os pares key-value são copiados com um fator de replicação de 3, o que na prática se traduz pelo armazenamento de cada par key-value em 3 nós de cluster diferentes. A motivação por detrás desta implementação é o facto de que, se o nó que originalmente armazena um par key-value ficar inativo, esse mesmo par torna-se indisponível. O verdadeiro número de cópias criadas pode variar entre 0 e 3, dependendo na realidade no número de nó ativos, para além do nó encarregue de guardar a cópia original.

Ao ser chamada a função *put*, é enviado o pedido para se armazenar o ficheiro em questão. É calculado o ângulo da *key* do ficheiro, que depois é usado para encontrar o nó que ficará responsável pelo ficheiro a guardar. Após este nó ter guardado o ficheiro, é executada esta lógica 3 mais vezes (se não houver mais 3 nós ativos para além do nó inicialmente responsável pelo armazenamento do ficheiro, a lógica repete-se o número de nós que ainda houver por usar no servidor). A cada iteração, retira-se da lista de nós elegíveis os nós que já contêm uma cópia do ficheiro.

5 - Concurrency

A implementação deste projeto também se apoia no uso de *thread-pools* de modo a permitir que um nó seja capaz de processar vários pedidos simultaneamente. Fazemos uso de um Executor que, por sua vez, cria uma “Fixed Thread Pool” com um número de threads igual ao número de cores da máquina que corre o programa.

```
//inicia o executor com uma thread pool com o nr de threads igual ao numero de cores do processador da maquina
exec = Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
```

```
node.getExecutor().execute(new MessageManager(this.node, content));
```

```
//Começa a correr o clusterManager responsável por aceitar conexões multicast do cluster do IP indicado
exec.execute(clusterManager);
```