

Second Assignment Report
CPD – LEIC

Distributed and Partitioned Key-Value Store

Grupo T08G01

Carolina Figueira - up201906845

Frederico Lopes – up201904580

Marta Mariz - up201907020

Índice

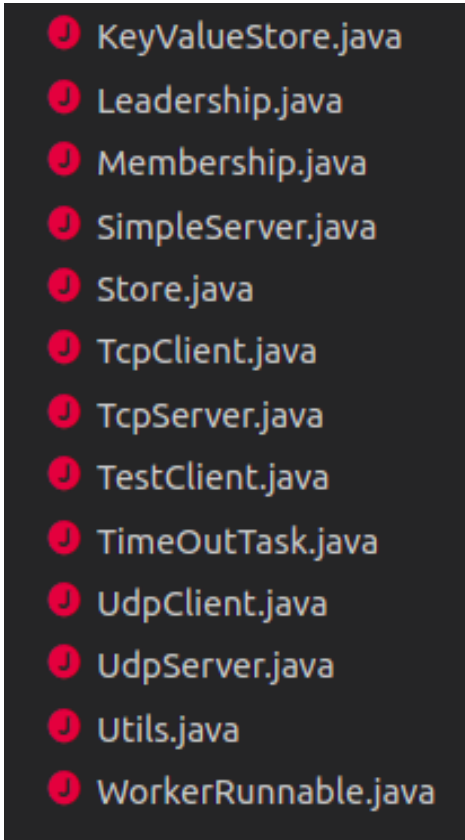
Índice	2
Introdução	3
Membership Service	4
Membership log	4
Leadership	5
Storage Service	6
Transferência de informação	7
Fault Tolerance	8
Concurrency	9
Thread-pools	9
Asynchronous I/O	10

Introdução

Este projeto corresponde a um sistema distribuído de armazenamento utilizando Key-Value hashing. O modelo apresentado pela proposta de trabalho é baseado na Store Amazon's Dynamo para um cluster grande. O sistema se baseia na utilização de chaves, obtidas através de uma função hash, que correspondem a nós e os seus ficheiros armazenados. Sendo assim, é um sistema organizado e partilhado, posto que o armazenamento de dado ficheiro depende do cálculo e da previsão de compatibilidade entre valores hash.

Sendo o objetivo deste trabalho partilhar o armazenamento de diversos ficheiros por uma gama de “nós”, deve ser garantida a persistência da informação independentemente de eventuais adversidades.

A estrutura do código está organizada nos seguintes ficheiros:



- 1 KeyValueStore.java
- 2 Leadership.java
- 3 Membership.java
- 4 SimpleServer.java
- 5 Store.java
- 6 TcpClient.java
- 7 TcpServer.java
- 8 TestClient.java
- 9 TimeOutTask.java
- 10 UdpClient.java
- 11 UdpServer.java
- 12 Utils.java
- 13 WorkerRunnable.java

Store.java: Contém a função main e recebe os argumentos para tratar da abertura das Stores

TestClient.java: Contém a função main e recebe os argumentos para enviar as operações ao cluster

KeyValueStore.java: Contém todas as funções da Interface KeyValueStore. É responsável pela gestão de ficheiros.

Membership.java: Contém todas as funções importantes para o funcionamento do Membership Service, incluindo as operações join e leave

Tcp/Udp Client/Server/WorkerRunnable .java: Classes que tratam das ligações TCP e UDP

Utils.java: Contém funções importantes para o gerenciamento e criação de hashes, assim como para o redirecionamento da informação armazenada

SimpleServer.java: Servidor TCP que recebe uma só ligação e depois fecha. Utilizado pelo TestClient e pelas funções de redirecionamento para receber a resposta do nó.

Membership Service

Para garantir que o cluster esteja sempre ciente e atualizado sobre os nós que estão ativos e pertencem ao sistema, é necessário que haja um serviço que segue um protocolo específico e se autorregula com alguma frequência. Para tal efeito, o membership service implementa conceitos importantes para garantir que todos os nós do cluster tenham informação atualizada e confiável.

Como proposto, um nó que entra no cluster após ter recebido join do TestClient abre uma ligação TCP numa nova porta, dedicada apenas ao join. A seguir envia três mensagens UDP indicando a nova porta onde está à espera de receber ligações TCP com as informações atualizadas. Para isto foi criada uma nova classe sendUdpJoin que reenvia a mensagem UDP de 3 em 3 segundos, enquanto não forem recebidas 3 respostas. No fim de reenviar três vezes é assumido que não há mais nós no cluster e fecha-se a ligação TCP.

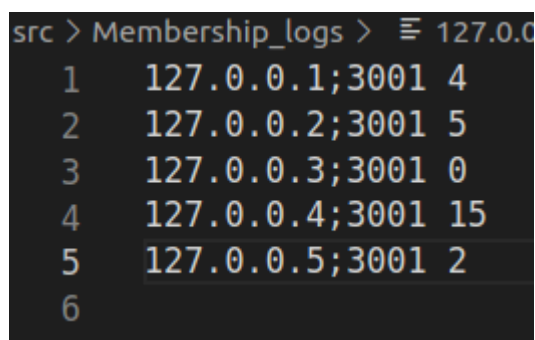
A mensagem UDP tem formato “cluster_join:<node_ip>:<store_port>:<counter>”.

Sempre que um node recebe uma mensagem UDP atualiza a sua membership log e visão dos nós ativos no cluster. A seguir envia a informação atualizada que corresponde aos membros da lista que o nó mantém com os ids, que tem formato “<ip>;<porta>”, de todos os nós ativos e as últimas 32 entradas nos logs.

A mensagem TCP tem formato “cluster_join_info:<node_ip>:<<node_ip>;<port>/ para cada nó ativo>:<<node_ip>;<port>:<counter>” para cada entrada no membership log>.

Quando um nó faz leave manda uma mensagem UDP a todos os outros a avisar a saída. O formato desta mensagem é “cluster_leave:<node_ip><port><counter>”.

Membership log



```
src > Membership_logs > ≡ 127.0.0.1
1 127.0.0.1;3001 4
2 127.0.0.2;3001 5
3 127.0.0.3;3001 0
4 127.0.0.4;3001 15
5 127.0.0.5;3001 2
6
```

O membership log mantém-se sempre atualizado e com o formato “<node_ip>;<node_port><counter>”. Possui uma linha por id, ou seja ip e port.

Leadership

Para uma melhor atualização da informação pelo cluster, é importante que exista um nó que garante estar sempre atualizado. Tal conceito de leadership, no nosso sistema, foi implementado de maneira que o líder é escolhido por ordem de chegada. O primeiro nó a juntar-se ao cluster torna-se líder. Para isso foi também criada a classe Leadership que envia de 1 em 1 segundo uma mensagem semelhante à enviada por TCP de um nó para o outro. Tal mensagem tem como objetivo atualizar os nós do cluster, portanto estes ao receberem tal mensagem devem atualizar o seu log, caso necessário.

A mudança de liderança acontece apenas quando o nó líder decide sair do cluster. O cálculo para determinar o novo líder funciona com base na ordem do TreeMap que guarda as hashes e ids. A primeira hash encontrada no TreeMap corresponde a um nó específico, e este se tornará o novo líder.

Storage Service

Depois do nó ser criado e de fazer parte do membership, é possível utilizar as funções da Interface KeyValueStore para realizar as operações de ficheiros pretendidas. As funções, que são chamadas remotamente pelo TestClient a partir de ligações TCP, são as seguintes:

- put(key, value), em que o TestClient envia a hash (key) a partir do conteúdo do ficheiro (value) e envia ambos para o nó. O ficheiro é criado e uma confirmação é retornada por TCP.
- get(key) em que o TestClient envia a hash retribuída pelo nó quando o ficheiro foi guardado. O conteúdo do ficheiro é retornado por TCP
- delete(key) em que o TestClient envia a hash retribuída pelo nó quando o ficheiro foi guardado. O ficheiro é apagado e uma confirmação é retornada por TCP.

Para cada nó é criada uma pasta cujo nome corresponde ao Id (uma string que representa o IP) onde todos os ficheiros correspondentes ao nó são guardados.

As funções da Interface KeyValueStore só devem ser chamadas por nós que façam parte do cluster visto que os nós fora deste não devem ter acesso aos ficheiros e podem ter informações desatualizadas quanto à informação dos ficheiros. Por esta razão, caso o nó não faça parte do cluster (o counter é ímpar), nenhuma função é chamada e é enviado uma mensagem ao TestClient a informar que o nó não pode ser utilizado para este tipo de operações.

```
if(arguments[0].equals("put") || arguments[0].equals("get") ||
arguments[0].equals("delete")){

    if(membership.getNodeCounter() % 2 != 0){

        //out of the cluster

        try {

            Thread.sleep(1000);

        } catch (InterruptedException e) {

            e.printStackTrace();

        }

        KeyValueStore.sendMessage("Node is out of the cluster,
operation not permitted");

        return;

    } (...)

}
```

Outro aspeto importante da implementação do KeyValueStore é o facto de que este utiliza consistent hashing para atribuir os ficheiros aos nós. Para identificar qual o nó que deve ficar com o ficheiro especificado é primeiro calculada a key do ficheiro (hash do conteúdo) que vai corresponder ao nome deste. Tendo isto, é de seguida necessário aceder à lista de nós ativos, que é um Treemap, cujas chaves são a hash do id do nó e os valores são o ip e a porta do nó concatenados numa só string do formato “ip;porta”. Uma função é responsável por determinar qual o nó com a hash mais próxima (em dúvida arredonda para cima) da hash do ficheiro. A complexidade temporal desta pesquisa é $\log(n)$.

```
if (!nodeMap.containsKey(hashCode)) {  
    target = nodeMap.ceilingKey(hashCode);  
    if (target == null && !nodeMap.isEmpty()) {  
        target = nodeMap.firstKey();  
    }  
}
```

Caso o ficheiro tenha sido enviado diretamente para o nó correto então o ficheiro é guardado diretamente. Caso o nó onde o ficheiro deva ser guardado seja diferente do atual então uma mensagem put, com os mesmo parâmetros que a enviada pelo TestClient, é enviada para o nó correto.

Transferência de informação

A cada operação join também é feita uma análise da compatibilidade da hash do novo nó com as keys já guardadas. Isto é, assim que um novo nó junta-se ao cluster, é calculada novamente a hash do nó mais compatível com a key que corresponde ao ficheiro; se tal nó for o novo é feita uma transferência da informação para este.

A mesma lógica foi utilizada em operações leave. É feita uma transferência de toda a informação guardada em um nó que está prestes a sair do cluster para os seus nós vizinhos - a partir do cálculo da compatibilidade das hashes utilizando um nodeMap que já não contém o nó que pretende sair. Assim, nenhuma informação é perdida durante o fluxo de entrada e saída de nós

Fault Tolerance

Apesar de alguns cenários de falha já terem sido referidos, outros cenários como um Ctrl+C não são mencionados ao longo das secções. Por isso, decidimos adicionar uma função de callback que é chamada sempre que a execução é terminada em resposta a um Interrupt do User, como um Ctrl+C, ou um evento do sistema como um logoff ou shutdown. Esta função queria uma Thread que executa a função leave, mandando assim os ficheiros para outros nós disponíveis e apagando os ficheiros localmente.

```
Runtime.getRuntime().addShutdownHook(new Thread() {  
  
    public void run() {  
  
        leave();  
  
    }  
  
});
```

Outro possível cenário de erro é o TestClient enviar os argumentos errados para o nó. Para dar a volta a isto, sempre que o TestClient envia uma mensagem da Interface KeyValueStore, este fica à espera duma resposta de confirmação. Caso a operação decorra como esperado esta vai ser uma mensagem de confirmação ou o conteúdo do ficheiro pretendido no caso do get, caso exista algum erro (como por exemplo o ficheiro que se pretende eliminar não existir), o TestClient recebe a informação de que a operação não foi concluída.

```
public void delete(String key) {  
  
    File file = new File( "../" + this.nodeId + "/" + key + ".txt");  
  
    if (file.delete()) {  
  
        sendMessage("Successfully deleted the file.");  
  
    }  
  
    else {  
  
        sendMessage("An error occurred.");  
  
    }  
  
}
```


Concurrency

Thread-pools

A parte central do nosso nó é o servidor TCP pois é este que recebe as mensagens e que, consoante as mensagens, executa a função suposta. Por isso, é importante que o servidor consiga sempre responder a novos pedidos e não fique preso na resposta anterior e por isso estamos a usar Threads.

O servidor TCP é lançado numa nova Thread que por sua vez cria um Thread Pool de 10 Threads para responder aos pedidos dos Clients.

Store.java:

```
TcpServer tcpServer = new TcpServer(node_id, store_port,
membership, false);

Thread threadTcp = new Thread(tcpServer);

threadTcp.start();
```

TcpServer.java:

```
ExecutorService threadPool = Executors.newFixedThreadPool(10);

(...)

this.threadPool.execute(new WorkerRunnable(clientSocket, "Thread
Pooled Server", nodeId, serverPort, membership));
```

Esta Thread pool possibilita que até 10 Threads possam estar a responder a pedidos ao mesmo tempo, o que nós consideramos suficiente visto que as respostas são praticamente instantâneas. Para ter a certeza que estas Threads não ficam presas a responder aos pedidos, estas só são responsáveis por ler a mensagem TCP e, de seguida, novas Threads são criadas para chamar a função adequada perante a mensagem recebida:

```
Utils callFunctions = new Utils(parsedInput, membership, kvStore);

new Thread(callFunctions).start();
```

Asynchronous I/O

Visto que cada Thread é responsável por uma dada operação, isto permite que estas terminem rapidamente e que os recursos sejam libertados para poder ser utilizados em outras operações. Um exemplo disto é o facto de que uma Thread pode ser responsável por ler a mensagem do TestClient e outra por ler/apagar/procurar o ficheiro e por fim por mandar uma resposta ao TestClient caso seja suposto como no caso do Get.

```
Utils callFunctions = new Utils(parsedInput, membership, kvStore);  
  
    new Thread(callFunctions).start();
```

Em cima podemos ver a Thread a ser criada para chamar a função pretendida, com base na mensagem recebida por TCP.

Um dos benefícios das Threads em Java é que mesmo que a Thread “pai” termine, a sua descendente continua a execução até ao final, permitindo assim terminar as Threads mal o seu papel esteja completado e permitindo assim uma grande divisão de recursos sem que o sistema operativo ou o processo principal seja sobrecarregado.

Assim, apesar de um número considerável de Threads ser criado, apenas algumas vão estar a correr concorrentemente.