Projeto 2

Key-Value Store com Computação Distribuída

Computação Paralela e Distribuida

Carolina Ferreira (up201905810)
Maria Miguel Ribeiro (up201906945)
Pedro Santos (up201907254)

Maio 2022

Índice

Introdução

No âmbito da cadeira de Computação Paralela e Distribuída foi desenvolvido um projeto de *persistent key-value store*, que desempenha a mesma função que um SSD ou um HD, para um cluster de grande dimensão. Para isso é necessário entender certos conceitos de computação distribuída, como *key-value store, multicasting, membership protocol, test-client* e *server*, bem como o funcionamento de uma ligação TCP IP/UDP.

O programa deve ser capaz de criar nós que se tornem servidores quando estes recebem um pedido de **join**, e deixem de ser servidores, quando recebem um pedido de **leave** por parte do cliente. Para além disso, o cliente pode pedir ao servidor que execute três operações, sendo estas: **put**, guardar um ficheiro com um par *key-value* atribuído, **get**, obter um determinado ficheiro a partir da chave desejada, e **delete**, apagar um determinado ficheiro guardado a partir da chave desejada. O protocolo deve permitir que haja vários servidores ativos, ao mesmo tempo, e que estes recebam vários pedidos em simultâneo. Para isso precisamos de evitar colisões, sendo importante atualizar a relação entre as chaves e os servidores responsáveis pelas mesmas, utilizando **consistent hashing**.

A linguagem utilizada durante todo o projeto foi Java.

Estrutura

Para facilitar a compreensão a divisão em *packages* foi feita da seguinte forma:

- **RMI:**
  - **RMIInterface:** este ficheiro é responsável por gerir as operações de join, que representa a ativação de um nó, passando este a ser um servidor, e leave, que trata de "desativar"o servidor, voltando este a ser apenas um nó.
- **TestClient:**
  - **TestClient:** este é o ficheiro de teste, que é utilizado para verificar se o serviço funciona da forma esperada.
- **Store:**
  - **AnswerRequest:** este ficheiro está encarregue de verificar todos os pedidos que o servidor recebe e enviar uma resposta válida para o cliente. As operações que tratam disso são:
    - **run():** verifica qual das operações foi pedida pelo cliente e, caso seja bem sucedido, evia uma mensagem, que varia dependendo a operação;
    - **hashing():** serve para encriptar uma determinada chave, e retorna essa chave encriptada;
    - **bytesToHex():** passa um array de bytes para uma chave encriptada do tipo string, é utilizado em complemento da operação hashing();
    - **put():** envia uma mensagem do tipo "Server response: put "+ key + "sucessfuly"ao cliente;

    * **get():** verifica se a chave pretendida existe. Caso encontre, envia uma mensagem do tipo "Server response: get key=<key>, get value=<value>". Se não existe, envia uma mensagem do tipo "Server response: get key=<key> does not exist!"ao cliente;
    * **delete():** envia ao cliente uma mensagem, dependendo de se consegue, ou não, encontrar a chave que se pretende eliminar. Caso consiga, envia uma mensagem do tipo "Server response: delete key=<key>". Senão, envia "Server response: delete key=<key> does not exist".

- **Hashing:**
  - **ConsistentHashing:** este ficheiro trata de todas as operações que estejam relacionadas com o anel de hashing, essas são:
    * **addNode():** adiciona um novo nó no anel de hashing
    * **removeNode():** remove um determinado nó do anel de hashing
    * **chooseNode():** escolhe um determinado nó do anel de hashing

Membership Service

Conceitos Importantes

O Membership Service tem a função de assegurar se as ligações entre os vários servidores e se as ligações entre o servidor e o cliente, funcionam devidamente. Cada servidor deve ter conhecimento de todos os servidores pertencentes ao cluster e por que pares (chave, valor) é que cada um é responsável. Para além disso, cada operação deve ser reencaminhada para mais do que um servidor.

A técnica utilizada para garantir que, caso haja alguma falha, o serviço continua a correr sem problemas, é a replicação, e o seu objetivo é guardar cada uma das três réplicas do valor recebido, em três servidores diferentes.

A segunda técnica utilizada foi o consistent hashing, que opera independentemente do número de servidores que se encontram ativos no cluster. É utilizado principalmente em serviços com um tráfego de informação muito elevado. Mas por que não utilizamos apenas *distribuited hashing*? No caso de um dos servidores crashar, as chaves pertencentes a esse servidor precisam de ser redistribuídas, e aí começa o problema do *distribuited hashing* pois, em vez de apenas redistribuir as chaves pertencentes ao servidor, redistribui todas as chaves. Isto pode gerar cach misses, já que as chaves podem não estar imediatamente inseridas na nova localização na tabela de *hashing*. Já o *consistent hashing* apenas redistribui as chaves atribuídas ao servidor que 'crashou'. O funcionamento do consistent hashing é o seguinte:

1. Criação de um anel de hashing;
2. Adição de cada nó (servidor) no anel, caso receba um pedido de 'join';
3. Pesquisa de qual dos nós é que está responsável por uma determinada chave, feita de forma sequencial;

4. Eliminação de um nó, caso receba um pedido de 'leave'.

Agora que temos uma noção dos conceitos importantes para um bom funcionamento do serviço, podemos passar à implementação do mesmo. Para um nó ser criado, são passados os seguintes argumentos:

$ java Store <IP Broad> <Port1> <IP Serv> <Port2>

O <IP Broad> corresponde ao IP de Broadcast, e é do tipo 224.0.0.X (X:[0,254]), o <Port1> corresponde ao número da porta desse IP, o <IP Serv> corresponde ao IP do servidor e neste caso é o 127.0.0.X (X:[0,254]) e o <Port2>, em que o servidor fica à escuta, lidos no comando de Store.

Para testar se o serviço funciona da forma esperada, utilizamos o comando:

$ java TestClient <node_ap> <operation> [<opnd>]

Neste comando passamos como argumentos o IP do servidor que o cliente pretende que realize a operação, e a operação respetiva (pode ser get, put, delete, join, leave) e de seguida no caso do put o próximo argumento será o nome do ficheiro que o cliente pretende guadar, no caso do get e do delete será a chave encriptada retornada pelo put.

Cada vez que um cliente pretende efetuar uma operação 'join' o nó entra no *cluster* tornando-se num servidor e é lhe associado um contador, inicialmente 0, que é incrementado em 1, sempre que o nó sai ou se junta ao *cluster*. Um contador par indica que o nó entrou no *cluster* e um contador impar significa que o nó saiu do *cluster*. Alguns dos membros do cluster enviam ao novo membro uma mensagem MEMBERSHIP e uma lista dos membros atuais do cluster através do protocolo TCP. Quando o cliente efetua um operação leave um nó sai do cluster deixando de ser servidor.

Implementação

Na implementação, existem dois ficheiros encarregues do membership service, o Store e o Node.

O Node verifica se o servidor está a escutar na porta certa e de seguida cria três threads que vão reservar um determinado core do processador para executar o request enviado pelo cliente. Depois indica qual a porta usada para broadcast, e cria um socket que envia essa informação para todos os servidores, o que permite que todos estejam atualizados quanto ao estado do cluster

No Store implementamos duas funções 'join' e 'leave'. A função join com o objetivo de inserir um nó no cluster começa por ler um ficheiro 'Cluster.txt' e incrementa o contador. A função leave tem o propósito de remover um nó servidor do cluster, dessa forma fizemos o mesmo que na função join mas desta vez com a ação de remover.

Storage Service

Conceitos Importantes

O serviço de storage tem a função de armazenar todos os dados importantes, como os pares (key, value) para isso implementamos uma tabela de hash. Utilizamos *consistent hashing*, que é uma técnica que permite redimensionar uma tabela de hash, ou seja, alterar o número de buckets, sem remapear todas as chaves da tabela já explicada anteriormente. Esta técnica é utilizada para tornar mais eficiente a gestão do anel de *hashing*.

A atribuição dos pares *key-value* aos nós é feita verificando o nó que contém o *hashed id* mais próximo do valor da chave.

Estes pares são armazenados em diretórios que são criados para cada nó servidor do *cluster* no qual o nome corresponde ao IP do servidor e o ficheiro contido representa um par, em que *key* corresponde ao nome do ficheiro, e *value* ao seu conteúdo.

Quando ocorre uma mudança no *cluster* os servidores podem necessitar de transferir chaves para outros servidores. No caso em que se verifica um *join* o sucessor do nó que entrou atribui-lhe as chaves que são menor ou igual ao seu *id*. Quando é efetuado um *leave* antes do servidor sair do cluster, atribui ao seu sucessor os pares *key-value* que lhe estava encarregue.

Implementação

Na implementação, o código que trata deste ponto está dividido em dois ficheiros diferentes, sendo estes: Operations e AnswerRequest.

O ficheiro **Operations** é responsável pela implementação de todas as operações que podem ser *requested* pelo cliente.

No ficheiro **AnswerRequest** implementamos a função **run()**, que funciona como uma main e tem como objetivo controlar todos os procedimentos que estejam ligados ao serviço de *storage*.

Primeiro é feita a verificação da existência de um ficheiro do tipo txt chamado Cluster e, caso não exista, cria-o. O seu conteúdo é uma lista de todos os servidores, ativos e não ativos, no cluster. Cada linha representa um determinado servidor, contendo informação essencial sobre ele. De seguida abre o ficheiro, verifica se o nó que foi chamado a realizar uma operação do tipo 'join' ou 'leave' já está inserido no ficheiro e, se estiver, incrementa em 1 o contador, que é o último argumento da linha, se ainda não estiver inserido, adiciona a informação pretencida depois da última linha. Desta forma é possível manter registo sobre todos os nós pertencentes ao cluster.

O próximo passo é a verificação da operação a realizar, pedida pelo cliente. Para isso foi criado uma série de *switch cases* que tratam exatamente dessa verificação e separam os procedimentos destinados a cada uma das operações, sendo estas o 'put', 'get' e 'delete'.

1. **put:** caso receba um 'put', verifica se o nome do ficheiro que foi inserido na linha de comandos é válido, retornando uma mensagem do tipo "Invalid File ", no caso de não o ser. Se aceitar todos os parametros, chama a função **put()**. Esta função também pertence ao ficheiro AnswerRequest e o que faz é:
   (a) criar uma pasta com o nome correspondente ao ip correspondente ao servidor que o cliente deseja que execute a operação, dentro dessa pasta, é inserido um ficheiro, em que o nome é a chave do par e o seu conteúdo é o value. Depois é enviada uma mensagem do tipo -Value file Created Sucessfully".
   (b) dar *scan* do ficheiro Cluster.txt e verificar se o ip do nó que o cliente deseja que execute a operação, se encontra na lista de servidores, e escolhe-o, juntamente os dois servidores seguintes no ficheiro. Se encontrar, cria um socket para cada um dos três servidores e envia a informação por esse socket.
   (c) retorna uma mensagem do tipo "Server response: <key> sucessfuly".
2. **delete:** caso receba um 'delete', chama a função **delete()**:
   (a) criar o file path como no ponto a).
   (b) verifica se o ficheiro de existe, em caso afirmativo elimina a chave e o ficheiro e imprime uma mensagem de êxito. Senão imprimi uma mensagem a indicar que o ficheiro não existe.
3. **get:** caso receba um 'get', chama a função get() pertencente ao mesmo ficheiro, e tem a função de:
   (a) criar o *file path* da mesma forma que no ponto a).
   (b) verificar se o *file path* existe e, caso exista, então guarda o value e dá *display*.
   (c) por fim, retorna uma mensagem do tipo "Server response: Value=<value>. "se for bem sucedido e "Server response: Value=<value> does not exist. "caso não seja bem sucedido.

Replication

A replicação é utilizada para que em casos de um servidor por alguma razão se desconectar do *cluster*, os pares *key-value* pertencentes não fiquem indisponíveis.

Para isso efetuamos 3 replicas para cada par e armazenamos as copias em 3 servidores diferentes do cluster, de forma a que não haja perda de informação. Tivemos em atenção a execução de operações em diferentes ordens em diferentes réplicas e à possibilidade de um nó perder uma operação de eliminação e mais tarde tentar replicar o par de chaves eliminado nas outras réplicas, e por isso utilizamos "*tombstones*"que é um par de chaves especial que funciona como um marcador, indicando que um par com a chave correspondente foi eliminado, assim em vez de eliminarmos o par, substituí-lo pela sua "*tombstone*".

Concurrency

Para que um nó seja capaz de processar vários pedidos ao mesmo tempo implementamos ***thread-pools*** dado que este método permite obter simultaneidade de execução.

Cada vez que ocorre um pedido é atribuída uma *thread* para tratar do mesmo, a sua função é alocar mais 3 processadores tornando possível a resolução síncrona de pedidos.

Testes

Como referido anteriormente, o ficheiro que trata dos testes é o **TestClient**. O TestClient gere todo o processo, verificando para que processo encaminhar, podendo ser de Membership Service ou de Storage Service. Também verifica se todos os argumentos da linha de comandos estão corretos, enviando mensagens a indicar se o input está correto ou não. Para além disso verifica qual a operação que está a ser requested.

Conclusão

O desenvolvimento deste projeto permitiu-nos perceber melhor o funcionamento de *persistent key-value store* para um *cluster* de grande dimensões e de uma ligação TCP IP/UDP, bem como alguns conceitos relacionados.

Para nós, este projeto foi desafiante, pois confrontámo-nos com algumas dificuldades na implementação que nos exigiu mais tempo a entender melhor os conceitos envolventes e a estruturar corretamente o programa, contudo conseguimos superar estes contratempos e obter os resultados esperados.

Embora as adversidades, concluímos que o grupo conseguiu elaborar o trabalho prático com sucesso.