Computação Paralela e Distribuída

Relatório 2: Armazenamento chaves-valor num sistema distribuído

António Ribeiro up201906761
Diogo Maia up201804974
Filipe Pinto up201907747

Computação Paralela e Distribuída **1**
- Overview 3
- Membership Service 3
- Service Invocation 4
- Storage Service 5
  - Implementação 5
  - Get 5
  - Put 5
  - Delete 6
  - Hashing e métodos úteis 6
- Replication 6
  - Réplicas no get, put e delete 7
  - Réplicas no Join 7
    - Motivo dos passos 6 e 7 7
  - Réplicas no Leave 8
- Concurrency 8
- Conclusão 9

Overview

Este projeto tem como objetivo desenvolver um armazenamento de pares (chave-valor) num sistema distribuído. Para isso usamos consistent hashing para distribuir os pares pelos diferentes nós do cluster.

Membership Service

O serviço de Membership permite que um nó entre e saia da vizinhança. Para permitir tal, cada nó tem dois métodos de alto nível, o *join*() e o *leave*(). Cada nó ao ser iniciado deve interpretar a informação que detém do cluster desde a última vez que lá esteve que se encontra num ficheiro criado por este: *membershiplog.txt*. A partir dessa informação, o nó consegue deduzir o *membership counter*, ou seja, um número superior ou igual a 0 que caso seja par indica que este está dentro do cluster e se for ímpar indica que se encontra fora deste. A ideia é que na primeira vez que o nó entra no cluster, este inicializa este contador a 0 e quando sair, incrementa-se um a este contador e assim sucessivamente.

A informação de cada nó será representada num sistema de ficheiros em que o nome será a hash do ip do respectivo nó. Cada diretório destes terá um ficheiro *membershiplog.txt* que terá a visão que este nó tem sobre o cluster representando em cada linha um nó e o seu contador da *membership*.

Para evitar que nós detenham informação desatualizada, de 15 em 15 segundos, cada nó considerado atualizado enviará a informação sobre o *membership* para todos. Esta informação é enviada através de multicast. A mensagem enviada seria do género:

membership false sender_ip:sender_port
body:
membership_logs

Ao receber uma mensagem deste género, o servidor tende a comparar os logs recebidos adicionando qualquer log de um nó que não tinha sido registado e alterando o registo de um nó para o maior *membership counter* registado.

Os nós que são capazes de dar multicast são os nós que ao dar join receberam no mínimo 3 respostas e os nós que recebem a mensagem *membership* de nós atualizados.

Service Invocation

Join:

Um nó, quando iniciado pelo primeira vez, não fará parte da membership, sendo o seu counter -1. De tal forma, o nó precisa de receber um pedido de join() do TestClient para entrar no cluster. Ao receber este pedido da TestClient, o nó especificado começa por enviar uma mensagem através de multicast a todos os outros nós da membership a avisar que quer entrar. Estes nós recebem a mensagem, atualizam a sua visão da membership e processam os seus dados de modo a preencher o novo nó. Enquanto isso, os nós que receberam o pedido acabam por enviar uma mensagem do tipo *membership* para o servidor TCP criado no ip do nó criado e numa porta definida por este.

O nó que se quer juntar fica à espera de mensagens de membership dos outros nós. Para impedir o caso de o nó ficar para sempre à espera de respostas, o nó reenvia a mensagem de join() até mais duas vezes e o intervalo de envio é 5 segundos. Para impedir o total de três envios, o nó precisa de receber três mensagens *membership* de três nós diferentes. Se no fim de três envios, o nó não receber qualquer informação, este inicializa-se como o primeiro nó do cluster.

Este processo é iniciado quando o servidor TCP do nó que quer entrar recebe a mensagem de *join* do TestClient. Ao receber esta mensagem, ele cria uma thread encarregado de responder a esta mensagem (MessageHandler) e esta thread será quem criará uma nova thread(MembershipProtocolJoin) capaz de enviar as mensagens de *join* pelo servidor Multicast, criar o servidor TCP na porta especifica (a porta decidida por nós é 7777) e esperar as respostas que serão tratadas por outros threads do tipo MessageHandler.

No caso de um nó *crashar* e este encontrar-se no cluster, ao reiniciá-lo, não será necessário este receber um pedido *join* para entrar no cluster sendo que este atualizar-se-á com mensagens *membership* periódicas e já fazia parte do mesmo.

Leave:

Um nó, quando dentro do cluster, ao receber um pedido de *leave* do TestClient, tende a executar o seguinte procedimento:

1. Primeiro modifica o próprio *membership counter* incrementando mais um
2. Envie uma mensagem do tipo *leave* para os restantes membros do cluster por multicast
3. Cada membro que receba essa mensagem vai atualizar o membership counter de tal nó
4. Por fim, o nó é responsável por enviar aos outros nós as suas keys e apagá-las da sua base de dados

Storage Service

Implementação

O serviço de *Key-Value Store* (*TcpServer* e *Store*), está implementado recorrendo a um servidor TCP, criado na instância de um dado nó. Este abre também uma *threadpool* (num Executor) que acolhe todos os pedidos de aceitação. A classe *MessageHandler* analisa o tipo de pedido (operação e protocolo) para que seja devidamente tratado por um método da *Store*.

O formato de mensagem escolhido foi o seguinte:

operação emissorTestClient IP Porta

body:
key do ficheiro
[conteúdo do ficheiro, no caso de ser um put]
end

Get

Na função *String get(String filekey, boolean isTestClient)*, vamos progressivamente alargando a zona de procura do ficheiro. Primeiro, tentamos o armazenamento local, de seguida (recorrendo a *Pair<String, Integer> getNearestNodeForKey(String filekey)),* tentamos o nó mais próximo da key.

Por fim, se esta conexão falhar tentamos todas as possíveis réplicas deste nó. Se tal não for possível, a mensagem de resposta ao pedido inclui uma string de erro.

No caso de não sermos contactados pelo *TestClient,* temos garantias de que outras Stores (incluindo as réplicas), foram contactadas, logo podemos simplesmente procurar pelo ficheiro, e retornar o seu conteúdo.

Put

A função *String put(String filekey, String value, boolean isTestClient)*, começa por obter tanto a *Store* mais próxima capaz de armazenar o ficheiro (que pode ser ela mesmo), bem como as suas réplicas. Caso ele esteja inserido em algum destes grupos, deve guardá-lo. Se o pedido for feito por outro nó, basta armazená-lo e retornar.

Caso seja a *Store* que comunica com o cliente, está encarregada de contactar os nó mais próximo e as suas réplicas, reproduzindo a mensagem de *Put,* para que o ficheiro coexista em pelo menos 3 *Stores* funcionais.

É importante referir que considerando as *tombstones*, se o put estiver a guardar o ficheiro e detetar que já existe (com o sufixo “.delete”, renomeia-o, poupando qualquer duplicação).

Delete

A função *public String delete(String filekey, boolean isTestClient)*, segue praticamente a ação de um put, alterando a operação.

Procura o nó mais próximo do ficheiro, e as suas réplicas. Caso se encontre num destes grupos, apaga o ficheiro do armazenamento local (isto é, coloca ".delete" no fim do seu nome).

Se estiver a ser contactado pelo *TestClient,* propaga a operação a todas as Stores, que calcula serem responsáveis pelo ficheiro (tendo em consideração, o retorno destes pedidos, para comunicar se falharam ou não), caso contrário, basta apagar e retornar de forma bem sucedida.

Hashing e métodos úteis

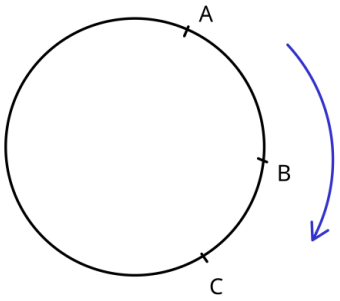
O hashing de uma *filekey* ou um IP (que serve como o ID da *Store*), é feito pela classe *ShaHasher*. Seguimos o modelo circular apresentado no enunciado do projeto (no sentido contrário aos ponteiros do relógio) e utilizamos *sha256*.

A ordem é estabelecida pela comparação das *strings* que resultam do *hash* e os nós contêm todo o intervalo de chaves de que são predecessores, até ao respectivo nó anterior.

- *Pair<String, Integer> getNearestNodeForKey(String filekey)*: função usada ao longo dos métodos da *Store*. Procura, recorrendo a uma binary search no cluster ordenado (e considerando apenas membership counters pares), o local mais próximo para a inserção de um ficheiro (ou um nó, no caso da hash igualar o seu ID).
- *String searchDirectory(String filekey)*: procura o ficheiro no diretório da Store e retorna o seu conteúdo se existir ou uma mensagem de erro.

Replication

Foi estipulado que as réplicas de um dado nó são os dois antecessores de uma dada *Store*, seguindo a ordem no sentido contra-relógio. De acordo com a imagem mostrada, o *Store* C é replicado por A e B (estes têm os ficheiros guardados pela *Store* C):



O uso de réplicas é retratado nos seguintes métodos:

- *List<Pair<String, Integer>> getStoreReplicas(String storeIP)*, dado um IP ordena o cluster e utiliza a pesquisa binária para encontrar a posição do mesmo na Store, armazenando as duas posições anteriores.

Réplicas no *get, put* e *delete*

- O comportamento das réplicas nestas ações foi abordado no capítulo anterior. Nestes casos servem ou como focos de alteração (*put/delete*) ou como possíveis *failsafes* para obter o valor de uma dada *key*.

Réplicas no *Join*

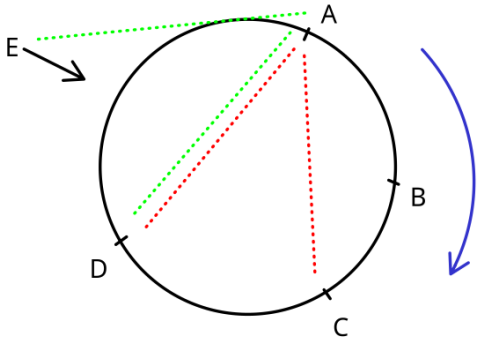
O handle de uma mensagem *Join*, recebida por UDP de um nó que acabou de entrar é feito da seguinte forma:

1. Guardamos os IDs dos ficheiros, cujos valores estão mais próximos da Store onde o pedido é recebido (antes de estabelecer a nova ordem do cluster).
2. Atualizamos a instância da store atual, através dos logs.
3. Enviamos a atualização do cluster para o nó que entrou (cumprindo os requisitos do protocolo membership).
4. Já tendo a lista de *Stores* atualizada, obtemos os predecessores e sucessores do novo membro.
5. Se a Store atual não se encontra em nenhum desses grupos, a função retorna, visto que nenhuma alteração nos afeta.
6. Se a *Store* for uma sucessora do novo nó, envia todos os ficheiros que previamente lhe pertenciam.
7. “Purgamos” todos os ficheiros que já não pertencem à Store (para cada um dos ficheiros que guardámos no passo 1, verificamos se ainda estamos na lista de preferência do mesmo - nó mais próximo + réplicas - se já não estivermos, apagamos).

Para o passo 7 usamos *private List<String> getPreferenceList(String filekey),* concatena a lista de predecessores e o nó mais próximo.

Motivo dos passos 6 e 7

A imagem ilustra o cenário em que dado um cluster [A, B, C, D], um novo membro E se junta ao mesmo, passando a lista para [A, B, C, D, E]. As linhas a vermelho e a verde representam as réplicas de A antes e após a inserção de E:



- Inicialmente, uma key em A estaria armazenada/replicada nas *Stores* D e C (os seus antecessores).
- Após a inserção de E, o nó A ficaria replicado em E e D. Por outro lado o nó E vai ocupar a posição de replicar A e B e armazenar todos os ficheiros em [E,D[.
- Podemos facilmente deduzir, que para atualizarmos o cluster de forma eficiente temos de garantir os seguintes passos:
    a. Os antecessores da nova Store (neste caso D e C) precisam apenas de apagar os ficheiros de que já não são responsáveis (D já não é uma réplica de B e C já não é uma réplica de A).
    b. Os seus sucessores, neste caso A e B, precisam não só de apagar os ficheiros que já não representam, mas também enviar os necessários para E.
    c. O nó A envia todos os ficheiros que guardava (incluídos no intervalo [A,D[), visto que E não só fica com o intervalo [E, D[, mas também replica [A, E[([A,E[∪ [E, D[= [A, D[).
    d. B envia apenas todos os ficheiros que lhe pertencem, já que E o tem de replicar.

Réplicas no *Leave*

- No caso de um leave, ocorre a situação contrária, em vez das outras *Stores* assumirem a responsabilidade de atualização o novo, a Store que sai tem de identificar os ficheiros que lhe pertencem (exceto os que replica), e envia-os para os membros do cluster respetivos.
- Tirámos partido do método *put* do sistema de *key-value*, propagando os ficheiros armazenados na rede. Invocando este método, a própria classe direciona-os para os destinatários corretos (com a *flag isTestClient* a falso, impedindo propagação no receptor).

Concurrency

- Nos dois tipos de servidores utilizados (tcp e udp), são instanciados objetos da classe Executor com uma threadpool de tamanho variável, que dado um pedido (aceitação de comunicação na socket), corre um *MessageHandler* (que implementa a classe *Runnable*).
- Desta forma, o processamento de pedidos diferentes é realizado em *threads* separadas, dentro do limite máximo de threads.

Conclusão

Este trabalho ajudou-nos a perceber a complexidade de sistemas distribuídos (key-value store), partindo do caso base de armazenamento para um sistema robusto e consistente.

Colocou em perspetiva o trabalho teórico desenvolvido neste tópico, bem como as diferentes formas de abordar falhas ao longo do “ecossistema” que poderá encontrar-se distribuído em pontos distantes de uma rede.

Por fim, realçou a importância de uma escolha adequada de protocolos de comunicação para diferentes funções, de pontos de replicação (visto que os nós são independentes e susceptíveis a falhas) e métodos de armazenamento.