Faculdade de Engenharia da Universidade do Porto

Serverless Backup Distributed Service

Autores:
Luís Martins
Carlos Freitas

Professor:
Pedro Nogueira

3 de Abril de 2018

1 Concorrência

Durante a implementação do trabalho foram implementadas várias técnicas com o objetivo de aumentar ao máximo a existência de concorrência. Uma vez que se trata de um serviço distríbuido, será natural a existência de tentativa de várias ações simultâneas. Como tal, existe interesse em atender o máximo pedidos possíveis em simultâneo para que a rede não fique demasiado sobrecarregada, e um cliente não tenha que esperar demasiado tempo para realização a ação que pretende.

Assim, o projeto foi implementado com base em threads, ou seja, o sistema tenta ao máximo libertar a main thread, criando novas para processar e enviar pedidos. Este tipo de estrutura é implementado com recurso à biblioteca ***java.util.concurrent.ThreadPoolExecutor***. Esta biblioteca permite que através da criação de um objeto **ThreadPool** existam várias threads a correr em simultâneo sendo estas geridas automaticamente. Deste modo, para os canais de comunicação existem 3 threads - MC, MDB, MDR - sendo que cada um destes canais recebe e envia as mensagens relativas ao seu tipo. Quando uma mensagem é recebida por cada um destes canais, é então criada uma outra thread para processar a mensagem recebida para que os canais fiquem logo livres para receber ou enviar novas mensagens.

```
1  MC = new MCSocket();
2   Runnable mcThread = MC;
3   this.exec.execute(mcThread);
4
5  MDB = new MDBSocket();
6   Runnable mdbThread = MDB;
7   this.exec.execute(mdbThread);
8
9  MDR = new MDRSocket();
10  Runnable mdrThread = MDR;
11  this.exec.execute(mdrThread);
```

No entanto, a criação de threads simples não permite que em momentos de espera(sleep) o sistema não fique bloqueado. Para tentar evitar este problema foram utilizadas ***Scheduled Threads***. Este tipo de threads permite agendar o lançamento de novas threads sem que para isso seja necessário manter recursos bloqueados durante o tempo que se pretende esperar. Este tipo de threads é utilizado principalmente quando é necessário processar mensagens

como o PUTCHUNK, ou o envio de mensagens CHUNK.

Foi também necessário considerar que, uma vez que o mesmo peer é acedido por várias threads, pode existir concorrência no acesso a variáveis globais do peer. Por esse motivo, as variáveis necessárias para serem consultadas são de tipos de dados que gerem concorrência, nomeadamente ***ConcurrentHashMap***. Este tipo de dados é utilizado nas estruturas existentes na classe StatusManager, responsável pela gestão dos dados mantidos por cada um dos peers.

```
1 ConcurrentHashMap<String,String> filesTables;
2 ConcurrentHashMap<String, Set<String>> deletedFiles;
3 ConcurrentHashMap<String,ChunkInfo> chunkTable;
```

Por fim, e uma vez que para a execução dos vários protocolos é necessário ler e escrever ficheiros, este tipo de ação foi implementada com recurso à biblioteca ***java.nio.channels.AsynchronousFileChannel***, permitindo assim que durante as leituras e escritas de e para ficheiros as threads não bloqueiem o sistema desnecessariamente, facilitamdo uma maior existência de concorrência. Nesta implementação é também considerada a criação de um objeto ***CompletionHandler*** utilizado para executar tarefas após o fim da leitura/escrita. Exemplo da utilização deste tipo de estrutura é a função saveChunks presente na classe FileManager que cria e guarda um ficheiro com os dados de um chunk, sem bloquear o sistema durante essa ação.

2 Enhancements

2.1 Backup

Para o serviço de backup base apenas se garante que os chunks criados são enviados para os outros peers existentes na rede e todos eles tentam guardá-lo. Ora uma vez que todos os peers guardam os chunks, o grau de replicação existente pode facilmente ser superior ao desejado, o que não será conveniente. Deste modo, este protocolo foi melhorado com o objetivo de tentar garantir ao máximo que o grau de replicação é o mais perto possível do desejado. Assim, foi criado em cada peer uma estrutura que, para cada chunk existente no sistema, mantém associado o grau de replicação desejado e o grau de replicação existente.

Sempre que um peer recebe uma mensagem PUTCHUNK analisa-a, e atualiza a variável referente ao grau de replicação desejado. De modo semelhante, sempre que um peer recebe uma mensagem STORED analisa-a e atualiza a variável referente ao grau de replicação existente.

Estas variáveis são mantidas globalmente em cada peer na estrutura **chunkTable** que associa a cada key (id do ficheiro + número do chunk) os respetivos valores de grau de replicação existente e grau de replicação desejado.

As mensagens STORED são recebidas e processadas por todos os peers, cada peer consegue facilmente saber qual o grau de replicação existente e o grau de replicação desejado de um certo chunk.

Assim, e uma vez que cada mensagem PUTCHUNK recebida é analisada com um delay entre 0 e 400 ms, quando esta for processada verifica-se se o grau de replicação existente é inferior ao grau de replicação desejado. Em caso afirmativo o peer guarda o chunk, rejeitando o caso contrário.

2.2 Restore

Uma vez que o restore básico apenas envia apenas uma mensagem GETCHUNK por cada chunk que pretende restaurar e como se trata de um protocolo de comunicação não fiável (UDP), em casos de ficheiros com bastantes chunks existe uma alta probabilidade de o restore não ser possível fazer.

Para tentar minimizar este problema, foi implementado um envio de mensagens com base em janela de congestionamento, garantindo que apenas estão em circulação no máximo 7 pedidos em cada momento. Deste modo, uma vez que existem menos pacotes em circulação, a probabilidade de erro é menor.

Não foi no entanto possível estabelecer comunicação TCP entre o peer que envia o chunk e o peer initiator, pelo que a alternativa encontrada tenta responder ao mesmo problema implementando no entanto uma solução não tão fiável e eficiente.

2.3 Delete

O serviço base de delete apenas envia uma mensagem informando que um certo ficheiro foi apagado. Ora uma vez que não existe qualquer confirmação de quem recebeu esta mensagem, e é possível que um peer não esteja na rede no momento em que esta mensagem é enviada, não existem garantias que o delete foi executado com sucesso.

Para tentar garantir que as mensagens DELETE chegam aos destinatários foi implementada uma resposta a esta mensagem. A nova mensagem DELETED é enviada por um peer que recebe uma mensagem DELETE e apaga algum chunk que tenha.

```
1 DELETED <VERSION> <SENDER_ID> <FILE_ID><CRLF><CRLF>
```

Por sua vez, o peer que pretende eliminar o ficheiro (obrigatoriamente o que anteriormente fez backup do mesmo), quando envia a mensagem DELETE também coloca na lista de ficheiros apagados, qual o fileId e os nomes peers que contém chunks do ficheiro apagado.

Deste modo, quando este peer recebe uma mensagem DELETED remove da lista de ficheiros apagados, o nome do peer que enviou esta mesma mensagem. Quando um fileId da lista, já não tiver nenhum peer associado este é removido da mesma.

Assim, para tentar garantir que os peers estão o mais atualizados possível, sempre que um peer se liga à rede, envia uma mensagem ALIVE por cada ficheiro distinto que contenha.

```
1 ALIVE <VERSION> <SENDER_ID> <FILE_ID><CRLF><CRLF>
```

Um peer ao receber uma mensagem ALIVE verifica se contém esse ficheiro na sua lista de ficheiros apagados. Em caso afirmativo, este reenvia a mensagem DELETE referente a esse mesmo ficheiro, para que o peer que não está atualizado se atualize, e aguarda novamente a chegada de uma resposta DELETED.