Sistemas Distribuídos

Projeto 1 – Serviço de backup peer-to-peer

Turma 7 – Grupo 09

Bernardo Almeida – up201806581

Caio Nogueira – up201806218

Conteúdo

Melhoria do subprotocolo BACKUP

A melhoria do subprotocolo **BACKUP** tem como objetivo evitar que o espaço em disco dos *peers* se esgote rapidamente.

Deste modo, caso a versão do protocolo passada através da mensagem ***PUTCHUNK*** seja **2.0**, o programa verifica se o *replication degree* real do *chunk* em questão é igual ou superior ao desejado. Caso seja esse o caso, o *chunk* não é guardado em disco no *peer.*

A verificação é feita através de um ***ConcurrentHashMap***, pertencente ao objeto *LocalStorage,* que faz corresponder a cada *chunk (key),* um *HashSet* (*value*) com o conjunto de *peers* que contêm esse mesmo *chunk* guardado em memória não volátil.

A implementação desta melhoria ao subprotocolo BACKUP permite uma ocupação muito mais eficiente do espaço disponível para cada *peer.* A versão não melhorada do subprotocolo, por não fazer uso deste recurso, guarda na maioria das vezes os *chunks* em todos os *peers* que recebem a mensagem de *PUTCHUNK.*

Verificação que permite a gestão do espaço de cada *Peer* (*BackupMessageManager.java*)

Melhoria do subprotocolo DELETE

A melhoria do subprotocolo **DELETE** tem como objetivo permitir apagar informação relativamente a ficheiros em *peers,* mesmo que estes não se encontrem em execução quando o subprotocolo é invocado pelo *initiator peer.*

A implementação desta melhoria implicou a adição de um novo tipo de mensagem, do tipo "***ACTIVE***". Deste modo, quando o *peer* é executado, este envia uma mensagem do tipo "*ACTIVE*", avisando os restantes que se encontra operacional e pronto a remover ficheiros do seu *filesystem.* É de notar também que o envio desta mensagem é feito após um período de espera que varia entre 0 e 400 milissegundos, de modo a evitar o envio simultâneo deste tipo de mensagens para o canal de ***multicast* de controlo**.

Para além de um novo tipo de mensagem, é necessário que os *peers* que invocam o subprotocolo DELETE guardem uma lista dos ficheiros que foram apagados durante a sua execução, de modo a poderem repetir as mensagens "**DELETE**" aquando da receção da mensagem "*ACTIVE*". Tal como acontece com as mensagens "*ACTIVE*", as mensagens "DELETE" que são enviadas para os *peers* que iniciam a sua execução esperam um período entre 0 e 400 milissegundos, de modo a evitar o bloqueio do canal.

Concorrência entre protocolos

Durante o desenvolvimento do programa, foi tida em conta a possibilidade de executar protocolos simultaneamente, tendo sido usadas algumas ferramentas disponibilizadas pela API de concorrência do JAVA, que permite a implementação da concorrência entre processos.

Estrutura geral do programa

A classe *Peer* possui o atributo *workers* do tipo ***ScheduledThreadPoolExecutor**,* ao qual foi atribuído um *corePoolSize* de 128. Este atributo permite a execução paralela de tarefas, que implementam a interface *Runnable,* através dos métodos *execute() ou Schedule()*, para execuções instantâneas ou agendadas, respetivamente. No contexto do projeto, estas tarefas são as classes do tipo *Task (PutChunkTask, GetChunkTask, ChunkTask, …),* que se encarregam do envio das respetivas mensagens, e as classes do tipo Message*Manager* (*RemovedMessageManager*, *BackupMessageManager*, *DeleteMessageManager*, …).

Estrutura dos canais *multicast*

A comunicação necessária para a correta implementação dos subprotocolos é feita através dos canais de *multicast* de controlo, *backup* e *restore*, que são acedidos através de 3 diferentes instâncias da classe ***MulticastChannel**,* sendo estes executados em paralelo através do membro *workers* da classe *Peer.* Deste modo, garantimos que o conteúdo enviado para estes canais será lido sem qualquer interrupção.

A classe *MulticastChannel* possui dois métodos: *run()* e sendMessage(). No método *run(),* o conteúdo do canal é lido em um ciclo infinito, sendo que, a partir do momento em que é recebida qualquer mensagem, o método *receive()* deixa de bloquear a execução do método e a mensagem recebida é interpretada por outra *thread* executada paralelamente. Deste modo, a *thread* responsável pela leitura do canal não

executa o processamento da mesma, de modo a proceder rapidamente para a próxima iteração do ciclo, ou seja, a próxima mensagem a receber.

```
workers.execute(controlMC);
workers.execute(backupMC);
workers.execute(restoreMC);
```

Execução paralela dos 3 canais de *multicast*

Envio e receção de mensagens

De modo a melhorar a eficiência no envio e receção de mensagens através dos canais de *multicast*, foi implementada uma classe *Message,* que recebe como argumento um *array* de *bytes* e faz a respetiva interpretação, com base no subprotocolo à qual essa mensagem pertence.

O envio das mensagens é feito através de objetos da classe ***MessageForwarder,*** que envia as mensagens para o canal correto. A receção das mensagens é feita através de objetos da classe ***MessageReceiver****.*

A implementação das classes referidas nos dois parágrafos anteriores permite que a interpretação das mensagens não seja feita nos respetivos canais de *multicast,* e que a receção de mensagens seja mais eficiente.

Existe, contudo, a possibilidade de algumas mensagens serem enviadas simultaneamente, visto se tratar de mensagens que são enviadas por diferentes *peers,* para o mesmo canal de *multicast* durante a execução de um mesmo subprotocolo*,* o que poderá levar ao bloqueio do programa. Para prevenir este tipo de situações, as mensagens que poderão ser enviadas em simultâneo esperam um período que varia entre 0 e 400 milissegundos.

Estruturas de dados utilizadas

As estruturas de dados utilizadas foram escolhidas tendo em conta as necessidades do programa, bem como o ambiente ***multithread*** que o caracteriza.

Deste modo, as estruturas de dados usadas neste sentido são maioritariamente ***ConcurrentHashMap's****,* que podem ser acedidos por diferentes threads e permitem leituras ao mapa de forma rápida e eficiente.

Para alem dos *ConcurrentHashMap's,* foi também utilizada uma ***SynchronizedList****,* de modo a garantir que é apenas enviada uma mensagem do tipo PUTCHUNK para cada *chunk* removido durante o subprotocolo RECLAIM.

O uso destas estruturas de dados garante um ambiente ***thread safe*** durante a execução do programa, permitindo que a informação aí contida possa ser acedida de forma segura por diferentes *threads* e que, consequentemente, a execução simultânea dos protocolos seja possível.