U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Projeto 1 - Distributed Backup Service

Sistemas Distribuídos

2019/2020 – 2º Semestre
14 de abril de 2020

T2G05
Ana Filipa Campos Senra (up201704077)
Cláudia Inês da Costa Martins (up201704136)

Descrição da implementação da concorrência entre os protocolos

Para tornar possível que os subprotocolos corressem de forma concorrente foram adotadas diversas práticas.

Primeiramente, uma vez que este projeto constitui um ambiente em que várias *threads* correm simultaneamente, em vez de utilizar um *HashMap* como estrutura de dados para guardar as informações das comunicações entre os *peers* recorremos ao uso de ***ConcurrentHashMap***. Esta estrutura de dados, sugerida pelos docentes, é uma ótima solução uma vez que é uma estrutura *thread-safe* e que tem um ótimo desempenho em ambientes *multithread*. Na imagem abaixo é demonstrada a utilização desta estrutura na classe *Storage* onde são guardadas informações sobre, por exemplo os *chunks* que guardou no backup de um ficheiro em *storedChunks*, entre outros.

```
public class Storage implements Serializable {
    private int overallSpace;
    private int occupiedSpace;

    private final ConcurrentHashMap<Pair<String, Integer>, Chunk> storedChunks = new ConcurrentHashMap<>();

    private final ConcurrentHashMap<Pair<String, Integer>, Integer> chunksGlobalCounter = new ConcurrentHashMap<>();

    private final ConcurrentHashMap<String, FileInfo> backedUpFiles = new ConcurrentHashMap<>();

    private final ConcurrentHashMap<Pair<String, Integer>, byte[]> recoveredChunks = new ConcurrentHashMap<>();
```

Além disso, recorremos ao uso da classe ***Java.util.concurrent.ScheduledThreadPoolExecutor***, que ao contrário de *Thread.sleep()* utilizado por nós início do desenvolvimento deste projeto, permite agendar a execução das *threads* após um determinado tempo de atraso definido por nós. Esta classe revelou-se útil em diversas situações, como por exemplo no protocolo de backup na classe *PutChunkThread* em que era necessário instaurar um atraso entre as sucessivas tentativas de envio das mensagens de PUTCHUNK para garantir que, caso o grau de replicação já tivesse sido atingido, não se voltasse a mandar essas mensagens, como pode ser observado na imagem abaixo:

```
if (numStoredTimes < replicationDeg) {

    System.out.println("\t> File: " + this.fileId + " Chunk No: " + chunkNo);
    System.out.println("\t\tStored: " + numStoredTimes + " times");
    System.out.println("\t\tDesired Replication Degree: " + replicationDeg);
    System.out.println("\t\tAttempt: " + this.counter);
    System.out.println("\t\tNext Delay: " + this.delay);
    System.out.println();

    PeerClient.getExec().execute(new Thread(() -> PeerClient.getMDB().sendMessage(message)));

   if (this.counter < 5) {
      PeerClient.getExec().schedule( command: this, this.delay, TimeUnit.SECONDS);
```

Outro exemplo desta classe foi no protocolo de restore: depois de mandar as mensagens GETCHUNK o *peer* espera algum tempo antes de tentar começar a restaurar o ficheiro para garantir que todas as mensagens CHUNK tenham sido recebidas, mais uma vez visível na imagem abaixo:

```
//IF ALL THE CHUNKS OF THE FILE WERE SAVED, PEER SENDS REQUESTS TO RECOVER THEM
int numChunks = 0;
for(Map.Entry<Integer, BackUpChunk> fileInfoEntry : backedUpFiles.get(fileID).backedUpChunk.entrySet() ) {

    System.out.println(" > SENDING MESSAGE: " + version + " GETCHUNK " + senderId + " " + fileID + " " + fileInfoEntry.getKey());
    byte[] message = MessageFactory.createMessage(version,  messageType: "GETCHUNK", senderId, fileID, fileInfoEntry.getKey());

    PeerClient.getStorage().addPendingChunks(new Pair<>(fileID, fileInfoEntry.getKey()));

    PeerClient.getExec().execute(new Thread(() -> this.sendMessage(message)));


    numChunks++;

}

System.out.println(" > RESTORING FILE");
//RECOVERING THE ALL FILE AFTER REQUESTING THEIR CHUNKS
PeerClient.getExec().schedule(new RestoreFileThread(fileID, filepath, numChunks),  delay: 10, TimeUnit.SECONDS);
```

Foi, também, utilizado o recurso '*synchronized*' em algumas funções de modo a impedir problemas de concorrência.

A linguagem Java fornece a classe '*Serialized*', que permite a recriação do objeto em memória, foi implementada pela classe '*Storage'*. A classe '*Storage*' que consolida toda a informação do *peer* é a classe guardada em memória (*Storage.ser*) permitindo assim guardar o estado da aplicação *peer* de modo a retomar a execução do *peer* posteriormente.

Melhorias efetuadas aos protocolos

Protocolo de Backup

Foi efetuada uma melhoria ao protocolo Backup. Na versão base, o espaço de backup disponível pode esgotar rapidamente e causar muita atividade quando esse espaço estiver cheio.

Deste modo, para garantir o grau de replicação desejado e, ao mesmo tempo, conservar espaço de memória, foi aplicado um protocolo de melhoria. Neste protocolo, os *peers* só guardam um CHUNK se o grau de replicação ainda não foi atingido. Deste modo, cada *peer* mantém uma contagem de quantos *peers* possuem o CHUNK guardado. O *chunk* só será guardado neste novo *peer* se o grau de replicação não foi atingido ainda.

Foi instanciada uma variável auxiliar *chunksGlobalCounter*. Esta variável é um *ConcurrentHashMap*, que tem como chave um *Pair* com o id do ficheiro e o número do *chunk* respetivamente e como valor o número de *peers* que guardaram o CHUNK.

Quando um *peer* recebe um PUTCHUNK espera um tempo aleatório entre 0 a 400 milisegundos até guardar o ficheiro. Deste modo, existe a oportunidade para outros *peers* guardarem o CHUNK. Antes de guardar o ficheiro, é confirmado se o grau de replicação foi atingido. Se o tiver sido, aborta e descarta o CHUNK.

Esta solução revelou-se ser bastante eficiente, pois diminui drasticamente a probabilidade de um CHUNK ter o grau de replicação superior ao desejado.

Protocolo Restore

Foi efetuado uma melhoria ao protocolo Restore. A versão base, se os CHUNKs forem grandes, pois embora apenas um *peer* precisar de receber a informação, como estamos a usar um protocolo *multicast*, todos os peers recebem.

Deste modo, para evitar que grandes blocos de dados sejam transmitidos para vários *peers* desnecessariamente, foi aplicado um protocolo de melhoria. Neste protocolo, os blocos de dados são enviados apenas para o *peer* que pediu para restaurar o CHUNK. Para o fazer, foi introduzido um novo tipo de

mensagem PORT que tem como corpo da mensagem a porta e o nome do *host* que irão ser usados para a comunicação TCP. O *peer* que envia a mensagem PORT abre a ligação TCP e espera para que o recetor aceite a ligação. Após isto, a informação é enviada pela *socket*.

Este protocolo mostrou-se bastante eficiente, pois diminui drasticamente a quantidade de dados transportados pelo protocolo *multicast*.