

**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.

Uma imagem com captura de ecrã, portátil

Descrição gerada automaticamentePrimeiramente, 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. Foi, também, utilizado o recurso ‘synchronized’ em algumas funções de modo a impedir problemas de concorrência.

Uma imagem com captura de ecrã, telefone

Descrição gerada automaticamenteAlém 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:

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:

Uma imagem com captura de ecrã, monitor, ecrã, sentado

Descrição gerada automaticamente

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 nos nós 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 *chuncks* possuem o CHUNK guardado. O *chunk* só será guardado neste novo peer se o grau de replicação ainda foi atingido.

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 o *peer* recebe um PUTCHUNK espera um tempo aleatório entre 0 a 1 segundo até guardar o ficheiro. Foi escolhido este tempo, pois mostrou-se o tempo suficiente grande para permitir a receção das mensagens STORED. Deste modo, existe a oportunidade para outros peers guardarem o CHUNK. Antes de guardar o ficheiro, confirma 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, não é desejável, pois apenas um *peer* precisa de receber a informação do ficheiro, no entanto, como estamos a usar um protocolo *multicast*, vários 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 irá ser usada para a comunicação TCP. O *peer* que envia a mensagem PORT abre a liação 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.