Projeto 1 - Serviço de Backup Distribuído

Relatório Final

Sistemas Distribuídos

Mestrado Integrado em Engenharia Informática e Computação

12/04/2021

T1G09

Pedro Daniel Fernandes Ferreira - up201806506

Pedro Varandas da Costa Azevedo da Ponte - up201809694

Execução concorrente de protocolos

Para garantir a concorrência entre os diferentes protocolos, de modo a que um *peer* possa lidar com as diversas mensagens que são enviadas e recebidas simultaneamente e de forma a poder enviar e receber diversos *chunks* ao mesmo tempo, recorremos à implementação de *multithreading*.

Para atingir a concorrência, recorremos à classe da API do Java *java.util.concurrent.ScheduledThreadPoolExecutor*, que permite agendar um gestor de "tempo limite", sem usar qualquer *thread* antes que o tempo limite expire. Desta forma, evitamos recorrer ao uso de *Thread.sleep* que pode provocar a existência de demasiadas *threads* ao mesmo tempo, utilizando-se assim demasiados recursos e limitando-se a escalabilidade. Com a utilização desta classe, não necessitamos de nos preocupar com o número de *threads* existente e a correr simultaneamente, uma vez que este é limitado e é gerido API do Java.

De forma a garantirmos a concorrência entre as várias *threads* no acesso, de forma segura, às tabelas onde são guardadas as informações sobre os ficheiros dos quais foi feito *backup* por um determinado *peer*, *chunks* guardados, distribuição dos *chunks* pelos vários *peers* existentes, entre outras, recorremos à classe *java.util.concurrent.ConcurrentHashMap*.

Quando um *peer* é iniciado (“Peer.java”), criam-se três *threads* que são associadas aos três canais *multicast* (MC, MDB, MDR) que os *peers* utilizam para enviar e receber mensagens. Nos canais que estas *threads* contêm, é feita a receção das mensagens enviadas pelos *peers*. Quando é recebida uma mensagem (“ChannelController.java”), cria-se uma nova thread - ManageReceivedMessages - que é responsável por processar a mensagem recebida. Desta forma, permite-se que seja possível processar diversas mensagens ao mesmo tempo.

```java
public void run() {

    // maximum size of a chunk is 64KBytes (body)
    // header has at least 32 bytes (fileId) + version + messageType + senderId + chunkNo + replicationDegree
    // so 65KBytes should be sufficient to receive the message
    byte[] buf = new byte[65000];

    try {
        MulticastSocket multicastSocket = new MulticastSocket(port);
        multicastSocket.joinGroup(address);

        // listens multicast channel
        while(true) {
            // receive a packet
            DatagramPacket packet = new DatagramPacket(buf, buf.length);
            multicastSocket.receive(packet);

            byte[] received = Arrays.copyOf(buf, packet.getLength());
            ManageReceivedMessages manager = new ManageReceivedMessages(this.peer, received);

            // call a thread to execute the task
            this.peer.getThreadExec().execute(manager);
        }
    } catch (IOException e) {
        System.err.println(e.getMessage());
        e.printStackTrace();
    }
}
```

Para cada *thread* ManageReceivedMessages, verifica-se qual é o tipo da mensagem (PUTCHUNK, CHUNK, DELETE, …) e chama-se uma nova *thread* que vai ser responsável por executar a respetiva parte do subprotocolo.

```java
// checks the message type and then creates a new thread to treat that message
public void run() {
    // message: <Version> <MessageType> <SenderId> <FileId> <ChunkNo> <ReplicationDeg> <CRLF>
    String[] messageStr = new String(this.message).split(" ");
    // System.out.println("Manager message: " + messageStr);
    switch (messageStr[1]){
        case "PUTCHUNK":
            Random r = new Random();
            int low = 0;
            int high = 400;
            int result = r.nextInt(high-low) + low;
            this.peer.getThreadExec().schedule(new PutChunkMessageThread(this.message, this.peer), result, TimeUnit.MILLISECONDS);
            break;

        case "STORED":
            this.peer.getThreadExec().execute(new StoredMessageThread(this.message, this.peer));
            break;

        case "DELETE":
            this.peer.getThreadExec().execute(new DeleteMessageThread(this.message, this.peer));
            break;

        case "GETCHUNK":
            this.peer.getThreadExec().execute(new GetChunkMessageThread(this.message, this.peer));
            break;

        case "CHUNK":
            this.peer.getThreadExec().execute(new ChunkMessageThread(this.message, this.peer));
            break;

        case "REMOVED":
            this.peer.getThreadExec().execute(new RemovedMessageThread(this.message, this.peer));
            break;

        case "DELETED":
            this.peer.getThreadExec().execute(new DeletedMessageThread(this.message, this.peer));
            break;

        case "WORKING":
            this.peer.getThreadExec().execute(new WorkingMessageThread(this.message, this.peer));
            break;

        case "CHUNKTCP":
            this.peer.getThreadExec().execute(new ChunkTCPMessageThread(this.message, this.peer));
            break;

        default:
            break;
```

Dentro de cada subprotocolo, para cada mensagem que é necessário enviar, cria-se uma nova *thread* responsável por enviar a mensagem através do canal *multicast* correto - ThreadSendMessages. Deste modo, é possível enviar várias mensagens simultaneamente, garantindo-se assim a concorrência entre protocolos.

```
public class ThreadSendMessages implements Runnable {
    private byte[] message;
    private ChannelController channel;


    public ThreadSendMessages(ChannelController channel, byte[] message) {
        this.message = message;
        this.channel = channel;
    }


    @Override
    public void run() {
        this.channel.sendMessage(this.message);
    }


}
```

Na nossa implementação, retiramos também partido da sincronização em Java, tornando assim alguns métodos *synchronized*. Assim, quando uma *thread* está a executar um determinado método sincronizado, mais nenhuma poderá executá-lo ao mesmo tempo, ficando as outras bloqueadas até que a primeira termine a execução.

.

Enhancements

1. Backup Enhancement

Este *enhancement* tem como objetivo melhorar o subprotocolo de *Backup* original, de forma a garantir que a replicação atual de cada *chunk* é igual à replicação desejada, permitindo, assim, que haja mais espaço para fazer *backup* de ficheiros em comparação com a utilização do subprotocolo inicial.

Na versão inicial (“1.0”), o *initiator peer* envia uma mensagem PUTCHUNK para todos os *peers* ativos através do canal MDB. Cada *peer*, ao receber esta mensagem, tenta imediatamente guardar a informação recebida no *body* da mensagem, criando um *chunk* para tal. De seguida, espera um tempo aleatório entre 0 e 400 ms antes de enviar uma mensagem STORED através do canal MC. Todo este procedimento pode ser visto no ficheiro “PutChunkMessageThread.java”.

No caso da versão melhorada (“2.0”), invertemos a fase de espera do *peer*, ou seja, em vez de este esperar um tempo aleatório antes de enviar a mensagem STORED, espera-o quando recebe a mensagem PUTCHUNK (“ManageReceivedMessages.java”). Após esta espera, já no “PutChunkMessageThread.java”, verifica se o número de replicações do *chunk* que se deseja guardar já atingiu o valor desejado ou não. Para tal, recorre-se a uma ConcurrentHashMap (*chunksDistribution*) que contém, para cada *peerId*, uma ArrayList com os *chunks* já guardados por esse *peer*. Desta forma, para se obter a replicação atual, percorre-se a lista de cada um dos *peers* e retorna-se o número de vezes que se encontrou esse *chunk*. Caso a replicação atual já tenha atingido a replicação desejada, então o *peer* já não necessita de guardar uma cópia deste *chunk*. Caso contrário, o *peer* guarda uma cópia do *chunk* e adiciona o id do mesmo (*fileId_chunkNo*) à sua lista na ConcurrentHashMap (*chunksDistribution*).

Através da nova versão deste subprotocolo, conseguimos melhorar a versão inicial, uma vez que na grande maioria dos casos, a replicação do *chunk* irá ser igual à desejada, permitindo poupar espaço na memória dos *peers* e, assim, fazer *backup* de um maior número de ficheiros.

2. Restore Enhancement

No subprotocolo inicial de *Restore* (versão “1.0”), é usado o protocolo UDP para transmitir pacotes entre *peers* de forma a obter os *chunks* que pertencem a um determinado ficheiro, para que este possa ser recuperado. Contudo, o protocolo UDP é um protocolo que não garante a não perda de pacotes aquando da transmissão dos mesmos. Para além disso, como, na prática, apenas um *peer* necessita de receber os *chunks* e estamos a usar um canal de *multicast* para enviar todos os pedaços do ficheiro, podemos realizar algumas alterações para melhorar este subprotocolo.

Para resolver estes problemas, a transmissão de *chunks* por *multicast* foi substituída por transmissão através de um canal TCP. Para tal, na versão melhorada, cada *peer* cria um canal TCP na porta 6000 + *peerID*, garantindo que cada *peer* tem uma porta diferente e permitindo vários comandos *Restore* serem executados simultaneamente. Esta porta é criada pelo *peer* e o *serverSocket* criado pelo mesmo é enviado como argumento na criação do *TCPChannel* (“TCPChannel .java”).

Tal como na versão inicial, o *peer*, ao receber uma mensagem do tipo GETCHUNK, vai iniciar uma nova thread onde criará a mensagem de resposta com a informação do respetivo *chunk* pedido. No entanto, em vez de instanciar a classe “ThreadSendMessages”, irá criar um novo socket para enviar a mensagem para o canal TCP criado pelo *peer* que faz o pedido (classe “ThreadChunkMessage”). Este procedimento pode ser visualizado nos ficheiros “GetChunkMessageThread.java” e “ThreadChunkMessage.java”.

De forma a garantir que os outros *peers* sabem que certo *chunk* já foi enviado por outro *peer*, um novo tipo de mensagem foi criada: CHUNKTCP (<protocol_version> CHUNKTCP <SenderId> <FileId> <ChunkNo>). Esta mensagem é enviada a todos os *peers* ativos através do canal de *multicast* MC, evitando o envio repetido de informação.

Através da nova versão deste subprotocolo, conseguimos melhorar a versão inicial, uma vez que passamos a usar um canal TCP que, embora possa não ser tão rápido como um canal *multicast* UDP, é mais confiável e, com a sua utilização, apenas o *peer* desejado recebe os *chunks*, em vez de estes serem enviados para todos os *peers* ligados à rede de multicast.

3. Delete Enhancement

Inicialmente, este subprotocolo (versão “1.0”) apenas permitia apagar os *chunks* de um ficheiro caso os *peers* que os contêm estejam a correr no momento em que o *initiator peer* envia a mensagem DELETE. Caso contrário, os *peers* nunca irão conseguir eliminar esses *chunks*, já que o *initiator peer* não consegue saber quais os *peers* que os apagaram, pelo que o espaço por estes ocupado nunca será recuperado.

O objetivo da versão melhorada (“2.0”) é permitir que os *peers* que não estão ativos no momento em que é enviada a mensagem de DELETE possam apagar os chunks dos ficheiros que foram eliminados quando forem novamente iniciados. Para tal, implementamos dois novos tipos de mensagens nesta versão: a mensagem WORKING e a mensagem DELETED.

Na versão “2.0”, quando um peer recebe uma mensagem DELETE, depois de apagar os chunks que contém do ficheiro indicado na mensagem, envia como resposta uma mensagem do tipo DELETED (<Version> DELETED <SenderId> <InitiatorId> <FileId> <CRLF><CRLF>) através do canal MD, permitindo assim ao *initiator peer* saber que este *peer* apagou os chunks do ficheiro. O *initiator peer* tem uma ConcurrentHashMap (*filesDeleted*) associada, na qual guarda a relação entre os ids dos ficheiros que apagou e os *peers* que apagaram os *chunks* destes ficheiros. Desta forma, ao receber uma mensagem DELETED, adiciona o *peer* que a enviou à lista do respetivo ficheiro.

Quando um peer inicia funções, envia uma mensagem do tipo WORKING para todos os *peers* através do canal MC, para os avisar da sua existência. Estes, ao receberem a mensagem, verificam para cada ficheiro presente em *filesDeleted* se esse *peer* já removeu os *chunks* desse ficheiro e, caso não o tenha feito, verifica em *chunksDistribution* se o *peer* fez *backup* de *chunks* desse ficheiro. Caso tenha feito, então, vai enviar, para cada ficheiro que o *peer* contém, chunks que devem ser eliminados, 5 mensagens de DELETE separadas por 200 ms cada, para evitar casos em que alguma mensagem possa ser perdida, garantindo assim que os *peers* recebem as mensagens. De seguida, o *peer* deve apagar os respetivos *chunks*, enviando como resposta uma mensagem DELETED.

Através da implementação desta melhoria conseguimos garantir que os peers apagam os *chunks* de ficheiros que foram apagados mesmo que não estejam ativos no momento em que a mensagem DELETE foi enviada.