U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Relatório do Primeiro Trabalho Prático

Sistemas Distribuídos

MIEIC - 2018/2019

Grupo 03 Turma 5

Fernando Alves - up201605270
Sandro Campos - up201605947

Introdução

Este projeto foi desenvolvido no âmbito da Unidade Curricular de Sistemas Distribuídos, do 3º Ano do MIEIC.

Este relatório irá explicar detalhadamente os enhancements feitos nos protocolos de *backup*, *restore* e *delete.* Além disso, irá também descrever as medidas que adotamos para obtermos um *design* concorrente.

Concorrência

De forma a tirar partido da execução paralela dos diversos protocolos implementados, adotamos uma implementação concorrente.

Para tal, usamos um objeto *ScheduledExecutorService* chamado *threadpool*, que, para além de permitir executar *threads* tal como o *ExecutorService* (com o método *execute()*), também permite agendar (com o método *schedule()*) processos para iniciarem passado um determinado intervalo de tempo, evitando assim o uso de *Thread.sleep()*. Pretendemos evitar o uso deste método pois pode levar a muitos threads estarem parados em *"busy work"*, prejudicando assim a concorrência. Cada *Peer* tem a sua própria *threadpool*.

Este é um dos casos em que usamos o método *schedule()*, neste caso para agendar a resposta STORED a uma mensagem PUTCHUNK, com delay entre 0 e 400 ms:

```
if (storage.saveChunk(chunk, senderId)) {
        // send stored message
        String storedMsg = Message.mes_stored(version, Peer.getId(), fileId, chunkNum);
        MessageSender sender = new MessageSender("MC", storedMsg.getBytes()); // send
message through MC
        int delay = ThreadLocalRandom.current().nextInt(0, 400 + 1); // random delay
between 0 and 400ms
        Peer.getThreadPool().schedule(sender, delay, TimeUnit.MILLISECONDS);
}
```

Para além do método *schedule()*, o *ScheduledExecutorService* fornece também um método para correr *Runnables* em intervalos de tempo constantes, chamado *scheduleAtFixedRate()*. Usamos este método para, por exemplo, guardar em memória não volátil a nossa estrutura de dados que mantém o grau de replicação dos chunks, como veremos mais abaixo.

Cada *Peer*, ao ser criado, cria os 3 canais (MC, MDB, MDR), que são objetos da classe *Channel*. Esta classe implementa a classe *Runnable*, e o respetivo método *run()* é um ciclo infinito em que são lidas as mensagens dos respetivos canais (cujo endereço e porta são especificados como argumentos do construtor do *Peer*), e para cada *packet* recebido é chamado um *MessageHandler*, que é também um *thread* que é enviado para a *threadpool*. Esta arquitetura foi escolhida para impedir que o *Peer* bloqueie ao tentar ler dos canais, e também para evitar que bloqueie ao executar o *MessageHandler* permitindo assim que o *Peer* esteja a executar múltiplos protocolos em concorrência.

Com o intuito de se usar uma estrutura de dados para manter o grau de replicação dos *chunks*, criou-se um *ConcurrentHashmap* que permite acessos designados por *"thread-safe"*. Esta característica é essencial, considerando que múltiplos *threads* do mesmo *Peer* podem estar em execução simultânea e a manipular valores presentes nesse mesmo mapa. Assim

sendo, devido ao acesso sequencial, dois *threads* não poderão adulterar valores despropositadamente.

Procedeu-se de igual forma à serialização, para cada *Peer*, do objeto representativo do seu armazenamento interno, passando este a ser gravado em memória não volátil. Para o efeito recorreu-se ao método *scheduleWithFixedRate()* mencionado acima, para agendar as tarefas de escrita periódica, de 30 em 30 segundos. Ao usar esta abordagem não há possibilidade de bloqueio, e tem-se em consideração o modelo concorrente do sistema, uma vez que as *threads* são usadas e libertadas para que possam realizar outro tipo de tarefas. O ficheiro obtido através da serialização é, em caso de existência, carregado automaticamente pelo *Peer*, restaurando o historial de *backup* e *restore* de uma execução anterior. O excerto de código que representa este mecanismo é apresentado a seguir, e está integrado no corpo do método *main* do *Peer*.

```
// load possible saved storage
Utils.loadStorage();

// schedule storage serialization
threadpool.scheduleAtFixedRate(new Runnable() {
   @Override
   public void run() {
     Utils.saveStorage();
   }
}, 30, 30, TimeUnit.SECONDS);
```

Enhancements

Backup Enhancement

De acordo com o explicitado no enunciado para o protocolo *backup*, o armazenamento de um determinado chunk envolve um grau de replicação, que na versão sem *enhancements*, pode vir a ser superior ao desejado para um determinado ficheiro. Isto acontece devido ao facto de este valor representar um limite inferior e não um limite superior ao número de *chunks* replicados. Assim sendo, e apesar da melhoria efetuada a este protocolo não ter sido significativa, decidiu-se travar esse armazenamento excessivo de *chunks*, guardando-os apenas quando o seu grau de replicação ainda não tiver sido atingido (sabemos esta informação pelo número de mensagens STORED de diferentes *Peers* já recebidas para aquele *chunk*). O método *writeChunk()* implementa este conceito, que pode ser observado no excerto de código abaixo.

```
private void writeChunk(Chunk chunk, int peerId) {
    String chunkName = chunk.getName();

    int currReplicationDgr = getChunkRepDgr(chunkName);

    // if under replication degree
    if (currReplicationDgr < chunk.getDesiredRepDgr()) {
        storedChunks.add(chunk);
        // update current replication degree
        insertInReplicationMap(chunkName, peerId);
        // store chunk locally
        chunk.write();
        // decrease peer available space
        availableSpace -= chunk.getData().length;
    } else {
        System.out.println("Warning: max replication degree already reached!");
        return;
    }
}
```

Restore Enhancement

Como foi mencionado no enunciado, o protocolo de *restore* descrito não é ideal, porque todos os *Peers* estão a receber os *chunks* pedidos, quando na verdade apenas um deles precisa de receber essa informação. Para o evitar recorreu-se ao uso do protocolo de comunicação TCP/IP ao enviar os *chunks* para o *initiator-peer*, tornando *unicast* o processo

de transferência dos dados, e libertando o sistema de sobrecargas desnecessárias. Enquanto que o *initiator-peer* atua como servidor, os restantes representam os clientes do serviço. A porta escolhida para a comunicação, e através do qual o servidor escuta foi pré definida como sendo a 8090.

```
ServerSocket server = new ServerSocket(8090);
System.out.println("Waiting for a client ...");

Socket socket = server.accept();
server.close();
System.out.println("Client accepted");

// takes input from the client socket
DataInputStream in = new DataInputStream(new
BufferedInputStream(socket.getInputStream()));

byte[] finalMessage = new byte[65000];
byte[] buf = new byte[65000];

int counter = 0;
while (true) {
   int readbytes = in.read(buf);
   if (readbytes == -1)
        break;
   System.arraycopy(buf, 0, finalMessage, counter, readbytes);
   counter += readbytes;
}
```

Como vemos no excerto de código acima, o primeiro *Peer* a ser aceite pelo servidor, durante a chamada *server.accept()* irá ser o que envia o *chunk*, sendo que todos os restantes irão receber uma exceção, significando que já existe um cliente a realizar a transferência e portanto terminarão. O *Peer* que está encarregue de enviar o *chunk* faz então a escrita para o servidor, que o irá ler, como podemos observar pela chamada ao método *read()* de *DataInputStream*, e guardar. O processo de escrita para o servidor é muito simples e descrita no trecho de código seguinte.

```
socket = new Socket("localhost", 8090);
// sends output to the socket
out = new DataOutputStream(socket.getOutputStream());
// send chunk message
out.write(message);
// close connection
out.close();
socket.close();
```

Delete Enhancement

O enunciado descreve um cenário em que o protocolo de *delete* não é ideal, porque se um *Peer* que dá *backup* a *chunks* de um ficheiro não estiver ligado quando a mensagem DELETE é enviada, esses *chunks* nunca serão apagados. Para prevenir isto, a versão *enhanced* do *delete* inclui o envio de uma mensagem de resposta por parte do *Peer* que apaga os *chunks* requisitados, confirmando ao recetor o processamento desse mesmo pedido. O formato desta mensagem é o seguinte:

"DELETED " + peerVersion + " " + senderId + " " + fileId + " \r\n\r\n"

Por sua vez, e para garantir que os pedidos de DELETE eventualmente chegarão ao seu destino, o *initiator-peer* executa periodicamente o envio do pedido DELETE para os ficheiros cujo grau de replicação seja superior ou igual a 1 (significando que existem *Peers* que guardam *chunks* de ficheiros apagados e ainda não os tinham removido nem enviado a resposta DELETED correspondente). Esta tarefa é executada pela *thread MessageDeleteCycle*, que é agendada em intervalos de tempo constantes na *threadpool* do *Peer*, e que podemos observar no excerto de código que se segue.

```
public void run() {
      Storage storage = Peer.getStorage();
      for (FileManager file : storage.getDeletionFiles()) {
         for (Chunk chunk : file.getChunkList()) {
            String message = Message.mes_delete(Peer.getVersion(), Peer.getId(), chunk.getFileId());
            MessageSender sender = new MessageSender("MC", message.getBytes());
            Peer.getThreadPool().execute(sender);
         }
      }
}
```