U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Sistemas Distribuídos

Relatório do Primeiro Projeto

Mestrado Integrado em Engenharia Informática e Computação
(2 de abril de 2018)

Grupo T3G12

Bárbara Sofia Silva **up201505628**@fe.up.pt
Julieta Frade **up201506530**@fe.up.pt

Introdução

Este relatório tem como objetivo explicar detalhadamente a melhoria implementada no protocolo base do projeto: ***backup***. Assim como também, descrever o design escolhido que permite a execução simultânea de protocolos e explicar a sua implementação.

Por fim, o projeto foi desenvolvido no âmbito da unidade curricular de Sistemas Distribuídos.

Protocolo de Backup

Relativamente à melhoria do protocolo de Backup, esta tem como objetivo garantir o grau de replicação desejado e consequentemente, poupar memória.

A forma mais eficiente, que o grupo encontrou, de implementar este melhoramento foi instanciar em cada *peer* a classe **Storage**, que por sua vez contém estruturas de dados que auxiliam na análise e gestão da informação. Neste caso, é de salientar a tabela **storedOccurrences**, cuja chave é uma *string* da combinação do ID do ficheiro com o número do *chunk* e o valor é o número de ocorrências da mensagem *STORED*, ou seja, quantas vezes é que esta mensagem foi recebida em relação a um *chunk* específico.

Assim, a melhoria foi implementada invertendo a ordem de operações do protocolo, isto é, sempre que é recebida uma mensagem *PUTCHUNK*, o *peer* espera um tempo aleatório entre 0 a 400ms até começar a escrever o ficheiro. No entanto, antes de o fazer, consulta a tabela **storedOccurrences**, onde tem acesso ao grau de replicação atual do *chunk* em questão e verifica se este é maior ou igual ao desejado. Caso a condição seja verdadeira, o *peer* descarta o *chunk* e aborta a sua escrita, no caso de ser falsa, o *peer* atualiza a estrutura de dados, escreve o ficheiro e envia uma mensagem *STORED*, que por sua vez vai fazer com que todos os *peers* atualizem a sua tabela.

Em suma, esta solução revelou-se ser bastante eficiente, visto a probabilidade de o grau de replicação de um *chunk* ser superior ao desejado ser muito baixa.

Execução simultânea de protocolos

Relativamente ao design implementado que permite a execução simultânea de protocolos, o grupo teve em conta inúmeros fatores.

Começando pela escolha apropriada de estruturas de dados, no caso das tabelas, em vez da utilização de *HashMap*, optamos por uma estrutura alternativa, *ConcurrentHashMap*. Esta é adequada para ambientes de *multi-thread*, pois é mais segura, escalável e tem um excelente desempenho quando o número de *threads* de leitura ultrapassa o número de *threads* de escrita.

O uso de ***Thread.sleep()*** para *timeouts* pode levar a um grande número de *threads* coexistentes, e tendo em conta que cada uma requer alguns recursos, consequentemente, a escalabilidade do design será limitada. Por esta razão, sempre que é necessário implementar um *timeout* e de forma a não bloquear a *thread* atual, usufruímos da a classe *java.util.concurrent.ScheduledThreadPoolExecutor*, que permite agendar um gestor de "tempo limite", sem usar nenhuma *thread* antes que o tempo limite expire. Este método é utilizado em várias instâncias do código, como por exemplo, na classe **Peer**.

```
public void restore(String filepath) {
    String fileName = null;

    for (int i = 0; i < storage.getFiles().size(); i++) {
        if (storage.getFiles().get(i).getFile().getPath().equals(filepath)) {
            for (int j = 0; j < storage.getFiles().get(i).getChunks().size(); j++) {

                String header = "GETCHUNK " + version + " " + id + " " + storage.getFiles().get(i).getId() + " " + storage.getFiles().get(i).getChunks().get(j).getNr() + "\r\n\r\n";
                System.out.println("Sent GETCHUNK");

                storage.addWantedChunk(storage.getFiles().get(i).getId(), storage.getFiles().get(i).getChunks().get(j).getNr());
                fileName = storage.getFiles().get(i).getFile().getName();

                try {
                    SendMessageThread sendThread = new SendMessageThread(header.getBytes( charsetName: "US-ASCII"),  multicastType: "MC");

                    exec.execute(sendThread);

                } catch (UnsupportedEncodingException e) {
                    e.printStackTrace();
                }
            }
            Peer.getExec().schedule(new ManageRestoreThread(fileName),  delay: 10, TimeUnit.SECONDS);
        } else System.out.println("ERROR: File was never backed up.");
    }
}
```

Nesta função, após o *peer* enviar a mensagem *GETCHUNK* para cada *chunk*, começa uma *thread* após 10 segundos que analisa de já recebeu os *chunks* todos, e por sua vez, restaura o ficheiro.

A classe **Peer** tem um atributo por canal *multicast*: ***MC***, ***MDB*** e ***MDR***. No método **main** é executada uma *thread* para cada um dos canais, onde é feita a receção das mensagens. Esta arquitetura permite que exista apenas uma *thread* por canal *multicast*.

```
private static int id;
private static ChannelControl MC;
private static ChannelBackup MDB;
private static ChannelRestore MDR;
private static ScheduledThreadPoolExecutor exec;
private static Storage storage;
private static double version;

private Peer() {
    exec = (ScheduledThreadPoolExecutor) Executors.newScheduledThreadPool( corePoolSize: 250);
    MC = new ChannelControl();
    MDB = new ChannelBackup();
    MDR = new ChannelRestore();
}
```

Em cada canal, sempre que é recebida uma mensagem é criada uma *thread* que a processa, ou seja, é possível processar várias mensagens ao mesmo tempo. A *thread* responsável pelo processamento das mensagens é a classe **ReceivedMessagesManagerThread**.

Adicionalmente, o grupo tirou partido da sincronização em Java, que é a capacidade de controlar o acesso de múltiplas *threads* a qualquer recurso partilhado. Desta forma, foi utilizado *synchronized* em vários métodos, visto ser a melhor opção para permitir que apenas uma *thread* tenha acesso a um recurso partilhado de cada vez. Um exemplo da aplicação desta metodologia é no método **backup** na classe **Peer**.

A linguagem Java fornece um mecanismo, chamado **serialização de um objeto**, que consiste em um objeto poder ser representado por uma sequência de bytes que incluem os dados do mesmo, bem como informações sobre o seu tipo e dos seus dados armazenados.

Após um objeto serializado ter sido escrito num ficheiro, este pode ser lido a partir do mesmo e a serialização anulada, isto é, as informações de tipo e os bytes que representam o objeto, assim como os seus dados podem ser usados para recriar o objeto na memória. Este mecanismo foi fundamental para guardar um estado da aplicação e partir do mesmo, ainda, visto a informação estar toda consolidada no atributo *storage* da classe **Peer**, foi apenas necessário guardar o objeto **Storage** de cada *peer*. A serialização deste objeto é feita no método **serializeStorage**, e a extração no método **deserializeStorage.**