13 de abril 2019

Faculdade de Engenharia da Universidade do Porto

Sistema Distribuído de Backup

Sistemas Distribuídos

Projecto 1

César Pinho – up201604039

Rui Guedes – up201603854

Introdução

No âmbito da unidade curricular de Sistemas Distribuídos foi desenvolvido um projeto que consiste num serviço distribuído de backup, que permite guardar ficheiros em computadores remotos que são utilizados como servidores (*Peers*) e geri-los através de uma aplicação (*TestApp*). Neste sistema é possìvel efetuar diversas ações, tais como, o *backup* e *restore* de ficheiros, *delete* dos mesmos e até mesmo a gestão do espaço de disco de um dado *Peer*.

O presente relatório tem como principal objetivo a descrição, acompanhada da explicação, do design adotado para a simultaneidade de protocolos, bem como, da implementação efetuada para melhoria dos protocolos base de ***backup***, ***restore*** e ***delete***.

As melhorias descritas neste relatório compõe a versão 2.0 do protocolo base (1.0) implementado, isto é, o protocolo melhorado (2.0) executa as ações tendo em conta todas as melhorias desenvolvidos mantendo-se interoperável com a sua versão base, sendo assim necessário, para a utilização de tais melhorias, a inicialização dos *Peers* especificando a versão do protocolo a “2.0”.

Simultaneidade de Protocolos

A execução simultânea dos diversos protocolos consiste num dos factores cruciais no desenvolvimento deste projecto. O seu design foi concebido de modo a maximizar a eficiência de execução, isto é, permitir correr diversos protocolos simultaneamente. No entanto para que tal fosse possível foi necessário ter em consideração diversos factores.

O primeiro fator, e o mais crítico relativamente a este tipo de implementação, consiste no facto de o número de threads a serem criadas poder ser excessivo tornando todo o sistema ineficiente rapidamente. Assim de modo a evitar este problema recorreu-se à utilização da classe ***java.util.concurrent.ThreadPoolExecutor*** que permite fazer a gestão eficiente de inúmeras threads de diversas formas. Uma destas formas, a utilizada no presente projecto, consiste na imposição de um limite máximo de threads que podem ser executadas simultaneamente, sendo que uma vez alcançado este limite, as threads a serem executadas aguardam numa fila de espera e assim que possível o *executor* efetua a reutilização de uma das threads previamente criadas. Deste modo permite-se assim reduzir a utilização da função *Thread.sleep()* que pode levar a um grande número de threads coexistentes e ainda reduzir o *overhead* da criação e terminação de threads.

O segundo fator consiste na possibilidade de diversos protocolos estarem a correr simultaneamente. Face a esta necessidade optou-se por criar uma classe *Multicast* que permite a generalização dos canais multicast (*MC, MDB, MDR*). A cada um destes canais encontra-se associado um objecto descendente da classe ***java.util.concurrent.ThreadPoolExecutor*** que é responsável por gerir e executar as *Tasks* associadas a cada um destes canais.

```
  MC = new Multicast(MULTICAST.get("MC")[0], MULTICAST.get("MC")[1]);
MDB = new Multicast(MULTICAST.get("MDB")[0], MULTICAST.get("MDB")[1]);
MDR = new Multicast(MULTICAST.get("MDR")[0], MULTICAST.get("MDR")[1]);
```

Deste modo tornou-se necessário converter cada protocolo e respetivos subprotocolos em *Tasks,* isto é, classes que implementam as interfaces *Runnable* ou *Callable*, para que seja possível a atribuição de cada uma delas ao canal respectivo.

Para permitir esta simultaneidade de protocolos é necessário que cada canal de multicast execute uma tarefa correspondente à classe *Listener*.

```
MC.getExecutor().execute(new Listener(MC));
MDB.getExecutor().execute(new Listener(MDB));
MDR.getExecutor().execute(new Listener(MDR));
```

Esta tarefa é responsável por estar permanentemente a ouvir o canal respectivo, e aquando a recepção de um *packet* esta classe cria uma nova thread responsável por decifrar a mensagem recebida e agir em concordância.

Deste modo várias mensagens, correspondentes aos subprotocolos, podem ser processadas simultaneamente o que por sua vez permite a simultaneidade desejada para os protocolos.

O terceiro, e último, factor a ter em conta no desenvolvimento deste design foi, dado o número de threads em execução, garantir a sincronização do acesso a estruturas partilhadas entre as diversas threads, bem como a manipulação de ficheiros. Relativamente ao acesso a estruturas de dados partilhados optou-se pela criação de uma classe *Synchronized* que permite a criação de métodos personalizados de manipulação destas estruturas e que garante a sua manipulação de modo sincronizado através de *synchronized blocks*. Uma vez que estas estruturas são partilhadas por várias threads recorreu-se ao uso da palavra-chave *volatile* para permitir que as alterações efetuadas nas estruturas ficassem imediatamente visíveis para as restantes threads. Relativamente à manipulação de ficheiros optou-se por não utilizar a classe ***java.nio.channels.AsynchronousFileChannel,*** visto que, a implementação efetuada requer que a manipulação destes seja feita e concluída de modo a poder prosseguir com as restantes acções. Deste modo a utilização de *synchronized blocks* para a manipulação de ficheiros é suficiente e garante a eficiência desejada dos protocolos.

Protocolo de Backup

Face à implementação base do protocolo de *backup* e com o intuito de o melhorar, foi desenvolvida uma segunda versão (2.0) do protocolo onde é possível guardar os ficheiros de modo a garantir o grau de replicação desejado sem ambiguidade, isto é, sem que valor desejado para o grau de replicação seja excessivo, permitindo assim uma melhor gestão do espaço de cada *Peer*. Para tal foi criada uma estrutura auxiliar ***stored_messages*** que guarda para cada *file_id* uma subestrutura com os diversos *chunks* e o seu respectivo grau de replicação.

Para que o grau de replicação de um *chunk* não seja excedido é necessário que o *Peer* que recebe uma mensagem de *Putchunk* conheça o grau de replicação atual, de modo a concluir se deve ou não guardar a informação recebida. Para que tal seja possível, cada *peer* atualiza a estrutura anteriormente referida por cada mensagem *Stored* recebida. De forma a validar a informação, uma vez que esta é acedida simultaneamente por várias *threads*, a estrutura é incluída na classe ***Synchronized*** que contém métodos de acesso usando *synchronized blocks*.

A melhoria implementada consiste inicialmente na inversão da ordem da operação de *delay*, isto é, a espera de um tempo aleatório entre 0 e 400 ms. Conforme a descrição do protocolo este deve ser executado antes do envio de uma mensagem *Stored*, no entanto, com a melhoria este passa a ser executado aquando da receção de uma mensagem *Putchunk*, isto para permitir que determinados Peers não guardem desnecessariamente certas *chunks.* Após este *delay*, é verificado o grau de replicação atual do *chunk* recebido na mensagem *Putchunk*, isto recorrendo à estrutura ***stored_messages***. Caso o grau atual seja igual ou superior ao desejado, o *Peer* descarta a mensagem, caso contrário, o *chunk* é guardado e a mensagem *Stored* é posteriormente enviada.

Tal como esperado, o resultado final da aplicação desta melhoria traduziu-se no aumento da eficiência do sistema na medida que cada *chunk* apenas é replicado o número de vezes desejado independentemente do número de *Peers* existentes, permitindo assim uma melhor manutenção e gestão de memória.

Protocolo de Restore

Face à implementação base do protocolo de *restore* e com o intuito de o melhorar tornado-o mais rápido e eficaz, desenvolveu-se um mecanismo responsável por evitar que para cada *chunk* enviada face a uma mensagem *Getchunk,* esta seja apenas enviada ao *Peer* iniciador. Para tal e recorrendo ao protocolo TCP foi criada uma ligação cliente-servidor para o envio dos vários *chunks* por parte dos *Peers* recetores (clientes) para o *Peer* iniciador responsável pela invocação do protocolo *restore* (servidor). Esta ligação utiliza a porta 4444 para o estabelecimento da ligação e da transferência de dados. Para que seja possível um *Peer* estabelecer a conexão com o *Peer* servidor, o *body* da mensagem *Getchunk* nesta nova versão do protocolo (2.0), em vez de ser nulo, como descrito no protocolo base (1.0), contém o endereço IP do *Peer* iniciador, isto é, o endereço IP de quem envia a mensagem *Getchunk,* não havendo assim a necessidade da criação de novas mensagens mas sim a edição das já existentes.

GETCHUNK <Version> <SenderId> <FileId> <ChunkNo> <CRLF><CRLF><Body>

Assim, o *peer* que contêm um *chunk* a enviar entra em ciclo infinito até conseguir conectar-se com o servidor ou até receber, pelo canal de *multicast MDR*, uma mensagem idêntica à que pretende enviar, isto é, a mensagem *Chunk*. Caso a conexão seja estabelecida com o servidor, é enviada uma mensagem *Chunk* sem *body*, pelo *MDR,* para comunicar aos restantes *peers* que o envio do *chunk* já está a ser efetuado e de seguida é enviado o número do *chunk* e o seu conteúdo através da ligação TCP. Após o envio, o *Peer* cliente fecha a ligação, tornando possível a conexão por parte de outros *Peers* para envio de outros *chunks*.

Em suma, o resultado desta melhoria revelou-se bastante significativo, uma vez que o tempo de execução do protocolo *Restore* foi bastante reduzido, isto porque, cada *chunk* é apenas enviado a um único *Peer,* ao *Peer* iniciador, permitindo assim que o canal de *multicast MDR* não fique sobrecarregado dado que as mensagens que chegam a este canal são de menor comprimento, resultado do envio do chunk através da ligação TCP, o que faz com que a sua interpretação seja mais rápida e eficiente.

Protocolo de Delete

Face à implementação base do protocolo de *delete* e com o intuito de o melhorar foi desenvolvida uma melhoria cujo objectivo consiste na eliminação de ficheiros de um dado *Peer*, mesmo que este se encontre inativo no momento em que mensagens do protocolo de *Delete* são enviadas. Com este intuito, foi necessária a criação de uma nova mensagem de controlo que é enviada sempre que um *Peer* é inicializado.

DELETEDFILES <Version> <SenderId> <LastExecution> <CRLF><CRLF>

Esta mensagem é utilizada para informar os restantes Peers que um determinado Peer acabou de ser inicializado e que pretende determinar quais os ficheiros que foram eliminados do sistema desde a última data de terminação <*LastExecution*> do *Peer*.

Deste modo para conseguir satisfazer tais pedidos é necessário que cada *Peer* não só efetue o registo das mensagens correspondentes ao protocolo *delete* mas também efetue o registo da data da sua terminação.

Assim e, com o desenvolvimento desta melhoria, sempre que um *Peer* envia uma mensagem de *Delete*, este regista, no ficheiro *deleted_files.txt,* a data de envio da mensagem e o *file_id* do ficheiro a ser eliminado. Deste modo aquando a recepção da mensagem *DeleteFiles*, torna-se possível conseguir reconstruir a mensagem correspondente ao protocolo de *delete* caso a data presente no campo <*LastExecution*> seja anterior às datas de envio de mensagem de *delete* guardadas no ficheiro *deleted_files.txt.*

Assim sendo sempre que um *Peer* recebe uma mensagem de *DeletedFiles*, este irá ler o conteúdo do seu ficheiro *deleted_files.txt* e verificar se o momento de envio das mensagens de *Delete* é posterior ao momento da última execução do *Peer* que enviou a mensagem *DeletedFiles*. Em caso afirmativo, o primeiro volta a enviar a mensagem de *Delete* para que o emissor consiga eliminar os ficheiros cujo conteúdo já deveria ter sido apagado. No caso do recetor não ter qualquer registo de *delete* após a última data de execução do emissor, não é executada qualquer ação.

Em suma, tal como esperado, o resultado final permite manter o sistema limpo e sem resíduos de antigos ficheiros guardados mesmo perante a inatividade de determinados *Peers* aquando o envio das mensagens de *Delete.*