Universidade do Porto

Serverless Distributed Backup Service

Pedro Fernandes **up201603846**@fe.up.pt

Francisco Filipe **up201604601**@fe.up.pt

Unidade Curricular: Sistemas Distribuídos

Docente: Prof.Diana Guimarães

Turma: MIEIC02

Grupo: 8

14 de Abril de 2019

Conteúdo

1.1 Introdução

Este relatório foi desenvolvido no âmbito do primeiro trabalho prático da Unidade Curricular de Sistemas Distribuídos. O seu objetivo é clarificar alguns dos aspetos principais da nossa implementação, nomeadamente os seguintes:

- **Instruções de Compilação e Execução:** As instruções necessárias para compilar e executar corretamente o programa desenvolvido, tanto em Windows como em Linux. Também está presente nesta secção a descrição dos scripts implementados, que ajudam na demonstração do trabalho.
- **Execução Concorrente de Protocolos:** Descrição detalhada dos mecanismos e estruturas de dados utilizadas no desenvolvimento deste trabalho, que permitem a execução concorrente dos diferentes protocolos. Esta descrição é acompanhada de alguns excertos do código fonte para ajudar a compreender a implementação desenvolvida.
- **Melhorias Implementadas:** Descrição de cada uma das melhorias propostas, da solução pensada e da implementação desenvolvida.

1.2 Instruções de Compilação e Execução

1.2.1 Compilação

Para compilar o trabalho desenvolvido, tanto em Windows como em Linux, apenas é necessário mudar o diretório atual para a pasta */src* e executar o seguinte comando:

```
javac *.java
```

1.2.2 Execução

Para executar o trabalho são necessários seguir os 3 próximos passos:

Iniciar o Registo RMI

Visto que este trabalho implementa a interface *RemoteMethodInvocation* (RMI) de Java, para o testar corretamente é necessário ainda iniciar o registo RMI com:

```
//Linux
rmiregistry &

//Windows
start rmiregistry
```

Executar os Peers

O comando de execução dos *peers* é o seguinte:

```
java Peer <protocol_version> <peerID> <peer_ap> <MCaddress> <MCport>
    <MDBaddress> <MDBport> <MDRaddress> <MDRport>
```

- protocol_version: Versão do protocolo compatível com o *peer*.
- peerID: Identificador úncico de um *peer*.
- peer_ap: Ponto de acesso do objeto do RMI.
- MCadress: Endereço IP do canal multicast de controlo.
- MCport: Porta do canal multicast de controlo.
- MDBadress: Endereço IP do canal multicast de backup.
- MDBport: Porta do canal multicast de backup.
- MDRadress: Endereço IP do canal multicast de restore.
- MDRport: Porta do canal multicast de restore.

Os endereços IP multicast acima utilizados devem estar entre os valores 224.0.0.0 e 224.0.0.255 que são os endereços multicast reservados para redes locais.

Executar a TestApp

O comando de execução da *TestApp* é o seguinte:

```
java TestApp <peer_ap> <operation> <opnd_1> <opnd_2>
```

- preer_ap: Ponto de acesso do *peer*.
- operation: Protocolo a ser executado.
- opnd_1:
- opnd_2:

Uma vez mais, à exceção dos scripts, todos os comandos acima deverão ser executados na pasta */src*.

1.3 Execução Concorrente de Protocolos

De forma a garantir a execução concorrente de protocolos da melhor forma possível foram tomadas algumas medidas tendo em conta os seguintes fatores:

- Dentro de um protocolo há várias mensagens a serem enviadas ao mesmo tempo.
- Alguns protocolos requerem a gestão de recursos "partilhados"(recursos que apesar de serem instanciados cada um no seu *peer*, são iguais em todos estes e, portanto, têm de ser coerentes).
- Um *peer* pode ter que guardar/enviar vários chunks ao mesmo tempo.

O primeiro passo na implementação da concorrência foi a utilização de *threads*. Para tal decidimos utilizar a classe *ScheduledThreadPoolExecutor*. Esta classe facilita bastante a implementação de *multi-threading* uma vez que já implementa mecanismos de baixo nível e os disponibiliza a um alto nível. Um dos aspetos que nos levou a optar pelo uso desta classe foi o facto de esta reciclar *threads*. A criação e destruição de *threads* é um processo dispendioso e, portanto, uma funcionalidade destas aumenta substancialmente a performance do nosso traalho. Outro aspeto também muito importante a ter em consideração foi o facto de ser possível agendar a execução de novas *threads* sem comprometer a execução da *thread* atual. A outra alternativa a esta opção na nossa perspetiva seria o uso de *Thread.sleep()*, que resultaria numa pior performance uma vez que bloqueia a *thread* atual.

O desenho da implementação *multi-threading* foi de acordo com a seguinte estrutura:

- Quando um protocolo é invocado, o *initiator peer* faz a chamada à função responsável por esse protocolo.
- Em todos os protocolos (à exceção do protocolo STATE) é iniciada a *thread MesssageSenderThread* que, ao receber como parâmetros a mensagem a enviar, o canal por onde enviar e o peer que a envia, envia a mensagem para o canal respetivo. Visto que a cada nova mensagem é executada uma nova *thread*, é possível o envio concorrente de protocolos.
- Sempre que um peer recebe no seu canal uma mensagem, executa uma nova *thread* chamada *MessageReceiverThread*. Esta *thread* , recebe os mesmos parâmetros da anterior e é responsável por interpretar que tipo de mensagem foi recebida e executar uma nova *thread* de acordo com este tipo.
- Apesar de existirem muitas threads a serem executadas caso haja concorrência de protocolo (ou utilizando protocolos com ficheiros maiores), uma vez que se utiliza o *ScheduledThreadPoolExecutor* não há qualquer preocupação quanto a este valor uma vez que ele está limitado.

```
switch (messageType) {
case "PUTCHUNK":
   peer.getScheduler().schedule(new ReceivePutChunkThread(message,
       length, peer), interval, TimeUnit.MILLISECONDS);
   break;
case "STORED":
   peer.getScheduler().execute(new ReceiveStoredThread(message, length,
       peer));
   break;
case "GETCHUNK":
   peer.getScheduler().execute(new ReceiveGetChunkThread(message, peer));
   break;
case "CHUNK":
   peer.getScheduler().execute(new ReceiveChunkThread(message, length,
       peer));
   break;
case "DELETE":
   peer.getScheduler().execute(new ReceiveDeleteThread(message, peer));
   break;
case "ACKDELETE":
   peer.getScheduler().execute(new ReceiveAckDeleteThread(message,
       peer));
   break;
case "ANNOUNCE":
   peer.getScheduler().execute(new ReceiveAnnounceThread(message, peer));
   break;
case "REMOVED":
   peer.getScheduler().execute(new ReceiveRemovedThread(message, peer));
   break;
default:
   break;
}
```

Analisando o código fonte acima apresentado é possível verificar que para o tratamento de um chunk é executada uma nova thread, permitindo assim o processamento simultâneo de vários chunks.

Em relação às estruturas de dados utilizadas para guardar *chunks*, canais de comunicação e mensagens proveninentes de outros *peers*, optámos pela utilização de *ConcurrentHashMap* em vez de *HashMap*. A principal vantagem da primeira reside no facto de assegurar o correto funcionamento na presença de várias *threads*, algo que não é assegurado com a utilização de *HashMap*. Visto que este trabalho utiliza em grande parte o conceito de *multi-threading*, esta foi a escolha óbvia a fazer.

1.4 Melhorias Implementadas

Neste trabalho foram propostas melhorias aos protocolos BACKUP, RESTORE e DELETE. Todas elas foram implementadas com sucesso e, como tal, iremos descrever nos próximos parágrafos as soluções desenvolvidas de uma forma mais detalhada.

1.4.1 Protocolo Backup

Melhoria Proposta

Implementar uma mudança ao protocolo que previna o rápido consumo de espaço dos *peers*, diminue o registo de atividade quando estes estiverem cheios e assegure o *replication degree* desejado. Esta melhoria tem de operar de forma correta com o protocolo da versão *vanilla*.

Implementação

Na versão *vanilla* deste protocolo, apesar de ser garantido o *replication degree*, o número de *chunks* guardados nos *peers* do sistema pode ser superior a este valor, visto que ele guarda em todos os *peers* que estejam à escuta. Isto faz com que, caso haja 100 *peers* à escuta e um deles faça BACKUP de um ficheiro com *replication degree* de apenas 2, 99 *peers* vão guardar os *chunks* desse ficheiro sendo que 97 deles estão a realizar trabalho desnecessário.

Na melhoria deste protocolo utilizámos a nosso favor a classe *Storage* implementada, mais concretamente o *ConcurrentHashMap confirmationMessages*. Este tem como chave uma *string* que combina o ID do ficheiro com o número do *chunk* do mesmo. O valor atribuído a cada chave é um *ArrayList* de inteiros que representam o ID do *peer* que enviou a mensagem STORED desse mesmo chunk após a execução do protocolo BACKUP. Assim, para saber qual o *replication degree* atual de um *chunk* no sistem, apenas é necessário percorrer o mapa e, quando a chave combinar, obter o número de peers que guardou o *chunk* recorrendo para isso ao tamanho do *ArrayList* respetivo.

A implementação do protocolo *vanilla* não permitia a um *peer* conhecer quantos chunks já tinham sido guardados no sistema. Para contornar esta adversidade foram realizadas três alterações ao protocolo:

- Antes de executar a *thread* de tratamento de mensagens do tipo PUTCHUNK, é adicionado um *delay* aleatório entre 0 e 400ms.
- Antes de guardar um *chunk* é verificado o seu *replication degree* atual.
- O envio da mensagem STORED é realizado antes do *peer* guardar o *chunk* que recebeu e não após.

A primeira alteração foi implementada recorrendo a um método da interface *ScheduledExecutorService* que permite agendar a execução de uma *thread*. Como são vários os *peers* que vão estar a tratar, paralelamente, as suas mensagens PUTCHUNK, ao agendarmos a execução das *thread*, minimizamos a concorrência sem comprometer a performance.

Visto que é necessário garantir um determinado *replication degree*, antes de guardar um chunk verifica-se se o seu *replication degree* desejado já foi atingido. Se sim, aborta-se a escrita do ficheiro. O facto de a execução das threads ser agendada e a concorrência minimizada ajuda nesta verificação pois os valores do *replication degree* conseguem ser mais atuais do que seriam caso a execução fosse imediata.

A escrita de um *chunk* num peer consome algum tempo precioso. Se realizarmos esta operação antes de enviar a mensagem STORED, podemos ter muitos *peers* a guardar um *chunk* desnecessariamente pois fazem-no antes de atualizar o valor do *replication degree*. Ao enviar o STORED primeiro, estamos a atualizar este valor e, só depois, a guardar efetivamente o *chunk*, conseguindo manter o valor do *replication degree* o mais atual possível para todos os peers.

1.4.2 Protocolo Restore

Melhoria Proposta

Implementar uma mudança ao protocolo de forma a que apenas o *initiator peer* receba os chunks enviados pelos restantes peers. Esta implementação deve interoperar com a versão *vanilla* e também utilizar TCP.

Implementação

Na implementação deste protocolo na versão *vanilla* é utilizado um canal multicast para enviar os *chunks* pedidos pelo *initiator peer*. Isto significa que todos os *peers* recebem os chunks, sendo que se estes últimos forem de grande tamanho diminuem a performance dos *peers*. Para desenvolver uma melhoria que recebesse crédito total tivemos que pensar numa estratégia que resolvesse este problema utilizando TCP.

Tendo isto em consideração a melhoria desenvolvida baseou-se no seguinte:

1. Quando o *initator peer* executa uma chamada ao método *restore* (responsável pelo envio de todas as mensagens GETCHUNK, espera das mensagens CHUNK e restauro do ficheiro pedido), são iniciados:

   - Um socket de comunicação por onde os *peers* enviam os chunks pedidos.
   - Uma nova *thread* chamada *TCPChunkReceiverThread* que fica à escuta de novas mensagens enquanto o socket estiver aberto.
   - Um semáforo com o valor inicial igual ao número de *chunks* do ficheiro. Este semáforo foi implementado utilizando a classe *CountDownLatch*. Este semáforo é utilizado para controlar o número de *chunks* que falta restaurar.

   Esta mensagem GETCHUNK passou a enviar também o endereço IP do *initiator peer* e a porta do socket instanciado por este. Desta forma, os *peers* que efetuarem a comunicação via TCP conseguem estabelecer a comunicação com o *initator peer* através de um socket que utilize estes atributos como argumentos.

2. De seguida são enviadas todas as mensagens GETCHUNK de uma só vez e, através da chamada ao método *await* da classe *CountDownLatch*, é bloqueado o *initiator peer*. Este só retorna à sua execução assim que o semáforo chegar a 0 e, portanto, sejam restaurados todos os *chunks*.

3. Assim que uma mensagem GETCHUNK chega a um novo *peer*, se este tiver feito backup do *chunk* pedido, executa uma nova *thread* chamada *TCPChunkSenderThread* que vai enviar pelo canal TCP um inteiro que representa o número do *chunk* que pretende restaurar.

4. A *thread TCPChunkReceiverThread* ao receber o número do *chunk* verifica se este já foi previamente restaurado. De seguida envia uma resposta booleana ao *peer* que enviou o número do *chunk* indicando se pode ou não prosseguir com o envio do *chunk* em si.

5. Ao receber esta resposta, a *thread TCPChunkSenderThread* age de acordo com a mesma. Isto é, caso indique que não deve prosseguir com o restauro é terminada, senão envia pelo canal TCP a mensagem CHUNK contendo o *chunk* que o *initiator peer* pediu.

6. Por fim a *thread TCPChunkReceiverThread* ao receber a mensagem CHUNK, separa os seus parâmetros e guarda toda a informação que lhe for útil, isto é:

   - Atualiza no *ConcurrentHashMap chunkMessages* o número de mensagens do tipo CHUNK que já recebeu para um determinado *chunk* (utilizado para prevenir o excesso de mensagens do tipo CHUNK no *initiator peer*).
   - Guarda o conteúdo do *chunk* enviado num *ConcurrentHashMap* conhecido como *restoredFiles*. Este mapa, tal como na melhoria anterior, tem como chave uma *string* que combina o ID do ficheiro com o número do *chunk*. O valor atribuído a cada chave é o conteúdo em bytes desse *chunk*.
   - Assinala que restaurou mais um *chunk* atualizando o valor do semáforo.

7. Este processo é cíclico desde o ponto 2 ao ponto 5 e só termina quando o semáforo desbloquear o *initiator peer*. Quando tal acontecer é fechado o socket inicializado no ponto 1 e invocada uma função responsável pela criação do ficheiro no sistema.

8. Assim que esta última função terminar a sua execução, o *initiator peer* termina a sua execução, terminando assim o protocolo RESTOREENH.

1.4.3 Protocolo Delete

Melhoria Proposta

Implementar uma mudança ao protocolo de forma a que *peers* que tenham feito BACKUP a chunks de um ficheiro e, no momento de execução do protocolo DELETE deste ficheiro, estejam desligados também possam eliminar os seus chunks assim que estejam ativos.

Implementação

Numa primeira abordagem a ideia para esta implementação era criar um ficheiro em todos os *peers* chamado de tasks.txt com a informação necessária dos ficheiros a eliminar da sua classe *Storage*. Este ficheiro consistiría numa série de linhas em que cada uma teria a seguinte estrutura:

$$DELETE \quad < fileID > \tag{1.1}$$

Desta forma, quando um peer iniciasse a sua execução, ao percorrer este ficheiro eliminava os *chunks* respetivos. No entanto, foi apenas após testar nos computadores da faculdade que nos apercebemos que esta solução não está corretamente implementada pois só funciona localmente.

Desta forma optámos por uma outra abordagem que funcionasse em todos os casos e não apenas localmente. Para isso decidimos criar mais duas mensagens de comunicação entre *peers*:

- **ANNOUNCE:** Esta mensagem é executada sempre que um *peer* começa a sua execução. Desta forma todos os *peers* ativos conseguem saber quando um outro *peer* é iniciado.
- **ACKDELETE:** Esta mensagem é executada sempre que o protocolo DELETE é executado num *peer*, sinalizando aos restantes que um *peer* acabou de apagar todos os *chunks* de um ficheiro no seu sistema.

Com estas mensagens implementadas o novo protocolo foi de acordo com a seguinte estrutura:

1. Quando um *peer* implementa o protocolo DELETE envia uma mensagem para os restantes *peers* dizendo qual o ficheiro que pretende eliminar do sistema.
2. Os *peers* que estiverem ativos, ao receberem esta mensagem eliminam os *chunks* respetivos e, de seguida enviam uma mensagem do tipo ACKDELETE contendo o ID do ficheiro que acabaram de eliminar.
3. Esta mensagem é recebida por todos os *peers* que imediatamente executam a *thread* *ReceiveAckDeleteThread*. Esta *thread* está encarregue de atualizar os valores dos atributos *confirmationMessages* e *deleteAcks*. O primeiro consiste num mapa que a cada *chunk* de um ficheiro guarda os *peers* que o contêm. O segundo, também um mapa, guarda os *peers* que já eliminaram um determinado ficheiro. Desta forma, apenas é necessário eliminar os *peers* que executaram o protocolo DELETE do primeiro atributo e adicionar ao segundo.

4. Caso um *peer* esteja desligado no momento em que é invocado o protocolo DELETE, no momento em que este passar a estar ativo é enviada uma mensagem ANNOUNCE contendo o ID do *peer* que acabou de se ligar.

5. Os restantes *peers* ao receberem esta mensagem executam a *thread ReceiveAnnounceThread* que, recorrendo ao método *getTasks* verificam se o *peer* que se ligou tem *chunks* guardados que já foram eliminados. Este método retorna um *ArrayList<String>* contendo os IDs desses ficheiros.

6. Depois de obter os IDs dos ficheiros, através de um ciclo, é enviada uma mensagem DELETE que, ao ser recebida pelo *peer* que acabou de se ligar, faz com que este atualize o conteúdo da sua classe *Storage* eliminando os *chunks* necessários.