Universidade do Porto

Serverless Distributed Backup Service

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

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

Unidade Curricular: Sistemas Distribuídos

Docente: They see me rolin'

Turma: MIEIC02

Grupo: 8

14 de Abril de 2019

Conteúdo

0.1 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.

0.2 Instruções de Compilação e Execução

0.3 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:

- Multi-threading.
- Estruturas de dados utilizadas para guardar os *chunks*.

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 mesmo parâmetros da anterior e é responsável por interpretar que tipo de mensagem foi recebida e executar a thread respetiva.

```
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*.

0.4 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.

0.4.1 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.

0.4.2 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

0.4.3 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