Faculdade de Engenharia
da Universidade do Porto

Relatório

Redes de Computadores

Mestrado Integrado em Engenharia Informática e Computação

Serverless Distributed Backup Service

Afonso Jorge Ramos up201506239@fe.up.pt
Edgar Filipe Carneiro up201503784@fe.up.pt

Abril 2018

1 Backup Enhancement

O sobprotocolo base de backup pode ocupar espaço indesejado, devido à velocidade com o qual faz o backup, e desta maneira, causar demasiada atividade nos nós quando esse espaço ficar completamente ocupado. Assim, decidimos chegar a uma alternativa que assegurasse o *replication degree*, evitasse os problemas de ocupação de espaço e que fosse compatível com subprotocolos sem a melhoria do mesmo.

Para procedermos à melhoria deste protocolo, decidimos utilizar informação que já estava armazenada no peer através de uma variável ConcurrentHashMap da classe ChunksRecorder, que guarda para cada ficheiro, o seu fileID e outro ConcurrentHashMap com o número de cada chunk, bem como uma classe ChunkInfo, que guarda o *replication degree, chunkSize e peersStored*.

```
1 private ConcurrentHashMap<String, ConcurrentHashMap<Integer, ChunkInfo>>
      chunksRecord = new ConcurrentHashMap<>();
2
3 private ConcurrentHashMap<String, Integer> filesDesiredRD = new
    ConcurrentHashMap<>();
```

Passando à explicação da implementação, quando uma mensagem PUTCHUNK chega a um dos *Peers* em vez de executar as operações normais, isto é, atualizar as estruturas de dados, escrever o ficheiro para o disco e aguardar durante um período aleatório entre 0 e 400 ms (StoreAction), alterámos a ordem das operações, e, começámos por bloquear o peer durante o período aleatório entre 0 e 800 ms, e, após este intervalo de tempo, o *Peer* verifica, finalmente, se o número de mensagens STORED recebidas para aquele chunk, do ficheiro em causa, já é igual ou superior ao replication degree desejado. Para este *enhancement* escolhemos o período aleatório, para que, teoricamente, existisse tempo suficiente para processar as mensagens de outros *Peers* que pudessem estar a correr a versão 1.0. Na análise das mensagens STORED, se o replication degree atual já for superior, o peer descarta a chunk, mas, caso não o seja, o peer atualiza as suas estruturas de dados e escreve o chunk no disco. Esta solução fez com que pudéssemos alcançar excelentes resultados. No entanto, para que esta melhoria esteja a funcionar na sua totalidade é necessário iniciar todos os peers com a versão 2.0.

2 Delete Enhancement

O subprotocolo DELETE foi melhorado no sentido de prevenir que um *Peer* guarde no seu disco chunks de um ficheiro já apagado. Isto é, se um *Peer* que fez backup de certos chunks de um ficheiro, de um outro *Peer*, não está ativo no momento em que o ficheiro é apagado, este nunca vai receber as mensagens de DELETE enviadas, pelo que nunca se irá aperceber que está a ocupar espaço desnecessário no seu disco. Desta forma, a melhoria implementada baseou-se na adição de uma mensagem, CHECKDELETE, que é enviada pela ação **CheckDeleteAction**, que envia uma destas mensagens por cada ficheiro que tenha presente no seu disco, e contém o identificador de cada ficheiro. Por sua vez, os *Peers* que receberem esta mensagem irão responder com um DELETE no caso de o ficheiro estar presente no CopyOnWriteArrayList de algum dos *Peers*. Desta maneira, sempre que um *Peer* volta a estar online, é atualizado dos ficheiros que foram apagados durante o seu tempo offline.

```
1 private CopyOnWriteArrayList<String> deletedFiles = new
    CopyOnWriteArrayList<>();
```

3 Restore Enhancement

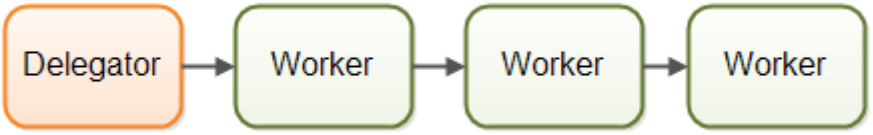
O subprotocolo de RESTORE pode tornar-se ineficiente quando se tratam de grandes chunks, com grande número de bytes, uma vez que, embora apenas um peer esteja à espera de receber esse chunk, como todas as mensagens são enviadas por *multicast*, a mensagem PUTCHUNK chega a todos os peers que se encontrarem ligados à rede, recebendo uma mensagem não desejada, desnecessariamente.

Assim, um peer com protocolo 2.0 pode estabelecer uma conexão TCP directamente com o peer que pediu esse chunk, desde que este também possua o mesmo protocolo. Desta forma, o peer que inicia o RESTORE, envia uma mensagem GETTCPIP, para que os Peers que tiverem acesso aos chunks do ficheiro pedido respondam com uma mensagem SETTCPIP, à qual vai associada o IP e a porta do Peer mencionado anteriormente. A partir daqui, imediatamente a seguir ao Peer, que possui os chunks para enviar, enviar a mensagem SETTCPIP, é criado, por ele, um servidor, ao qual o Peer que fez o pedido de restore se irá ligar após a interpretação da mensagem recebida com o IP e a porta. Infelizmente, não conseguimos implementar na totalidade esta funcionalidade, pelo que incluímos duas versões do protocolo melhorado, a versão normal e a versão 2. Na versão normal todo o protocolo está funcional, no entanto, apenas conseguimos estabelecer a transferência de 1 chunk, enquanto que na versão seguinte, tentámos, mas sem sucesso, replicar o anterior para múltiplos chunks, enviando previamente o número de chunks a enviar, para o cliente estar preparado.

De qualquer das maneiras, penso que demonstrámos ter capacidade de implementar esta funcionalidade, apenas teríamos de ter gerido melhor o tempo para o desenvolvimento deste projeto.

4 Concurrência

Ao longo do desenvolvimento deste projeto implementámos inúmeras técnicas com o objetivo final de possibilitar ao máximo a existência de concurrência, já que, afinal, se trata de um serviço distríbuido, será crucial a existência de ocorrência de várias ações em simultâneo. Desta forma, o máximo de pedidos têm de ser atendidos em simultâneo, para que não haja sobrecarga da implementação. Para além de que não é desejável que um Peer tenha de esperar para qualquer tipo de ação. Isto é, implementámos um modelo de concurrência comumente chamado de *assembly line concurrency model* ou *event driven systems*, tal como pode ser demonstrado pela seguinte imagem.



Assim, o projeto foi implementado com o objetivo de utilizar threads, ou seja, desenvolvemos o projeto com o objetivo máximo de tentativa de libertação da main thread, criando novas para processar e enviar os vários pedidos. Este tipo de estrutura foi implementado com ajuda e recurso às bibliotecas:

```
1 import java.util.concurrent.ThreadPoolExecutor;
2 import java.util.concurrent.ScheduledThreadPoolExecutor;
3 import java.util.concurrent.ConcurrentHashMap;
4 import java.util.concurrent.CopyOnWriteArrayList;
5 import java.util.concurrent.Executors;
6 import java.util.concurrent.ScheduledExecutorService;
7 import java.util.concurrent.ScheduledFuture;
8 import java.util.concurrent.TimeUnit;
9 import java.util.concurrent.atomic.AtomicLong;
```

Estas bibliotecas permitem inúmeras possibilidades, entre as quais, a criação de um objeto ThreadPool, onde podem existir várias threads a correr em simultâneo sendo estas geridas automaticamente. Deste modo, para os canais de comunicação existem 3 threads - Control Channel, Backup Channel e Restore Channel - onde cada um canal recebe e envia as mensagens apenas relativas ao seu tipo. Quando uma mensagem é

recebida por cada um destes canais, é então criada uma outra thread para processar a mensagem recebida para que os canais fiquem de imediato livres para receber ou enviar novas mensagens. Essa thread é criada usando a class MessageDispatcher que irá interpretar a mensagem recebida e decidir o que fazer com ela. Assim, o Dispatcher poderá gerar diversas ações sendo que estas, por sua vez, podem gerar outras threads ainda, dentro daquilo que é o funcionamento da ação em si. Assim, é possível distinguir quatro estrutras bem definidas responsáveis pela geração e manuseamento the Threads: A Peer principal, os Canais que lhe estão associados, O Dispatcher desencadeado pelos canais e as Ações resultantes.

No entanto, a criação de meras threads não permite a criação em intervalos de tempo de espera (sleep). Para tentar evitar este problema foram utilizadas Scheduled Threads. Este género de threads permite-nos agendar a criação de novas threads sem que para isso seja obrigatório reter recursos, bloqueando-os durante o tempo que é pretende esperar. Threads estas que foram utilizadas, por exemplo, no enhancement do BACKUP.

Para além de tudo isto, foi, também requirido considerar que, já que o Peer é acedido por várias threads em simultâneo, pode, também, existir concorrência no acesso a variáveis globais do peer. Devido a isto, as variáveis como ConcurrentHashMap, CopyOnWriteArrayList, AtomicLong, foram necessárias, para que pudessem ser consultadas. Estes tipos de dados foram úteis ao longo de todo o projeto, mas com mais enfâse nas estruturas existentes da classe ChunksRecorder, responsável pela gestão dos dados mantidos por cada um dos peers.