U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Faculdade de Engenharia da Universidade do Porto

Sistemas Distribuídos

3º ANO - **MIEIC**

Serverless Distributed Backup Service

Estudantes & Autores

Diogo Silva, up201706892

up201706892@fe.up.pt

João Henrique Luz, up201703782

up201703782@fe.up.pt

10 de **abril** de **2020**

sdis1920-t2g08

A. Introdução

No âmbito da unidade curricular de Sistemas Distribuídos foi-nos proposta a realização de um ***Serverless Distributed Backup Service*** utilizando ***Java***. Este relatório tem por objetivo documentar o *design* de concorrência implementado, assim como as melhorias efetuadas aos três subprotocolos sugeridos: ***backup***, ***restore*** e ***deletion***.

B. Subprotocolo de *Backup*

Neste protocolo, sem qualquer modificação, era inevitável que todos os *peers* "*online*", ou seja, subscritos ao grupo *multicast*, que recebessem uma mensagem de **PUTCHUNK** guardassem uma cópia desse *chunk*. Deste modo, mesmo que a *replication degree* fosse ultrapassada, nada impedia que houvesse desperdício de memória por estarem a ser guardadas mais cópias do que as desejadas. Para combater este problema foram criadas duas classes – **StoredChunks** e **StoredRecords** – usadas para haver registo das cópias de *chunks* já efetuadas por cada *peer* a partir de um **ConcurrentHashMap** (de forma a evitar *race conditions* entre *threads* de um *peer*). A primeira estrutura trata de guardar **ChunkInfo**'s[1] de *chunks* que estejam a ser guardados por esse mesmo *peer*, enquanto a última regista também **ChunkInfo**'s, mas de *chunks* já guardados por outros *peers*. Ambos os objetos implementam a interface **Serializable**, pelo que o seu conteúdo é mantido em memória não volátil, num ficheiro com extensão *.ser*.

Cada *peer*, sempre que recebe uma mensagem **STORED**, atualiza o seu registo de StoredChunks para, ao receber um **PUTCHUNK** e esperar um tempo aleatório de até 400ms, verificar se, entretanto, já alguém guardou o esse *chunk* e se a *replication degree* pretendida já foi atingida, evitando, assim, com bastante sucesso, o armazenamento repetido de *chunks* que já cumpram a *replication degree* desejada.

C. Subprotocolo de *Restore*

Tal como no protocolo anterior, acontecia haver vários *peer*s com os *chunks* pretendidos e estes não saberem se estariam a enviar um *chunk* inútil por um outro *peer* já o ter feito. Portanto, procedemos à implementação de uma nova classe: **RestoreRecord**, que mantém um registo numa **ConcurrentSkipListSet** (novamente para evitar problemas de concorrência) dos *chunks* que já foram restaurados. Isto acontece porque sempre que um *peer* encontra um *chunk* candidato a ser enviado, depois de esperar até 400ms verifica o seu registo de restauros (este último

[1] Classe genérica de **Chunk**, mas sem o seu corpo para evitar armazenar informação desnecessária.

atualizado sempre que o *peer* recebe uma mensagem de **CHUNK**) enviando o *chunk* apenas se já não tiver sido feito por outro.

Mesmo assim, esta solução não era eficiente *per se*, uma vez que, o restauro de um ficheiro para um dos *peers* implicava o envio dos *chunks* correspondentes pelo canal de *multicast* para todos os *peers* da rede, pelo que para ficheiros e *chunks* de tamanho elevado acabava por inundar o canal dos *chunks* não envolvidos com essas mensagens. Este problema é facilmente resolvido se ao invés de enviar os *chunks* do ficheiro por esse canal, estabelecermos uma ligação **TCP** direta entre o *peer* que tem o *chunk* pretendido e o *peer* que o deseja receber. Desta forma, quando um *peer* se apercebe que tem um desses *chunks* pretendidos, envia para o MC apenas uma mensagem a indicar o endereço IP e o *port* para estabelecer a referida conexão. Para tal, estabelecemos a mensagem **CONNECT**[2], em que o *hostname* é determinado descobrindo o IP *outbound* preferido do *peer* quando efetuada uma conexão UDP de teste e o *port* é o primeiro que se encontrar disponível (encontrado automaticamente pelo sistema) de forma a evitar a ocorrência de colisões entre *ports* utilizados.

D. Subprotocolo de *Deletion*

Por último surge a melhoria deste protocolo. Neste caso, acontecia que aquando do envio de um pedido de **DELETE** de ficheiro, caso os *peers* que estivessem a fazer *backup* de *chunks* desse ficheiro estivessem "*offline*", nunca viriam esse espaço de armazenamento livre. Para colmatar essa falha, optamos por implementar a mensagem **UPDATE**[3] enviada para o MC pelo *peer* sempre que este fica "*online*" na rede. Quando os outros *peers* recebem esta mensagem, tratam de enviar uma outra nova mensagem – **DELETIONS**[4] – para o MDR com uma lista das *deletions* efetuadas por estes.

Essa lista é enviada no `<Body>` da mensagem e é baseado no registo feito pela classe **RemoveRecord** (mantido em memória não volátil) que é atualizada sempre que uma mensagem de **DELETE** ou **BACKUP** é detetada, de forma a manter um equilíbrio entre os ficheiros que devem efetivamente estar ou não guardados.

Deste modo, o *peer* que acaba de se ligar tem a oportunidade de cruzar os seus registos com o dos outros *peers* e libertar ou não espaço desperdiçado.

---

[2] `<Version> CONNECT <SenderID> <FileID> <ChunkNo> <Hostname> <Port> <CRLF><CRLF>`
[3] `<Version> UPDATE <Placeholder> <Placeholder> <CRLF><CRLF>`
[4] `<Version> DELETIONS <SenderID> <Placeholder> <Placeholder> <Body>`

E. *Design* de Concorrência

O *design* de concorrência implementado suporta a execução simultânea de várias instâncias dos diferentes subprotocolos. Tal deve-se não só à forma como são recebidas e tratadas as mensagens enviadas pelos *peers* envolvidos, mas também às restrições impostas à escrita e leitura de informação por parte dos *threads* lançados.

Para que seja possível a receção simultânea de mensagens, foi criado um objeto da classe **MulticastManager**, que, uma vez inicializado, leva à execução de três objetos da classe **Runnable** em diferentes *threads*, cada um destes responsável por ler continuamente um dos três canais *multicast* necessários. Para a criação destes *threads*, recorremos a objetos da classe **ScheduledThreadPoolExecutor** e **ScheduledExecutorService**, que permitem o escalonamento das execuções. Estes objetos foram também úteis para evitar o uso excessivo da função **Thread.sleep()**, uma vez que esta pode levar a um consumo elevado de recursos, limitando a escalabilidade do *software* desenvolvido.

Para conseguir processar todas as mensagens recebidas em simultâneo, é, também, criado para cada mensagem, um novo *thread* encarregado de processar a mensagem e levar a cabo o subprotocolo associado à mesma.

No decorrer dos vários subprotocolos os *threads* acedem a informações guardadas em objetos das classes **ConcurrentHashMap** e **ConcurrentSkipListSet**. Estas estruturas de dados são *thread safe*, por outras palavras, oferecem uma implementação que garante de forma "segura" a execução de operações de inserção, remoção e leitura, impedindo a ocorrência de *race conditions*. Para que o mesmo ocorra em métodos nos quais são manipulados dados em estruturas não protegidas, os mesmos foram convertidos em métodos síncronos (*synchronized*). Deste modo, apenas um *thread* pode aceder a estes dados de cada vez, mantendo a informação coerente.

A implementação das medidas descritas acima, assegura a receção e processamento simultâneos de mensagens em canais diferentes, assim como a manipulação segura de dados por diferentes *threads*, tornando possível a execução simultânea de várias instâncias dos protocolos.