Project 1 - Distributed Backup Service

SDIS 2019/2020

Grupo T1G5:
Daniel Brandão - up201705812
Pedro Moás - up201705208

"Peer State"

Antes de explicar como certos problemas foram resolvidos, é necessário explicar o modo de representação do estado de cada peer.
Essencialmente, são estes os campos principais do estado do peer, e são os únicos que são guardados em memória não volátil (periodicamente), pois o seu conteúdo não pode ser perdido após um crash do sistema.

```
private final FileBackupLog backupState;
private final StoredChunkLog storedState;
private int capacity;
```

FileBackupLog é, essencialmente, um ConcurrentHashMap<String, FileBackupEntry>, com *helper functions* específicas para esta aplicação. Do mesmo modo, StoredChunkLog é essencialmente um ConcurrentHashMap<String, StoredChunkEntry>.
FileBackupEntry guarda uma entrada de ficheiros cujo backup foi iniciado pelo próprio peer, enquanto StoredChunkEntry guarda uma entrada de chunk que foi guardado pelo próprio peer. Os seus campos estão demonstrados, respetivamente nas duas imagens abaixo.

Campos de FileBackupEntry:

```
private final String pathName;
private int desiredRepDeg;
private final int numChunks;
private Boolean waitingForDeletion = false;
private final List<Set<Integer>> storers;
```

Campos de StoredChunkEntry:

```
private final int desiredRepDeg;
private final int chunkSize;
private final Set<Integer> storers;
```

"Protocol Enhancements"

1. Backup

O enhancement do chunk backup protocol visa evitar a rápida ocupação do espaço de backup dos peers e a grande atividade dos nodes cujo espaço de backup está completo.

Cada peer, ao receber uma mensagem ***PutChunk*** vai verificar se o chunk já foi guardado nesse peer verificando no ***storedLog*** do peer utilizando ***hasChunk(ChunkID***

chunkID). Para além disso, se o peer não tiver espaço para guardar o chunk nem prossegue à fase seguinte. De seguida, o peer vai esperar aleatoriamente entre um número proporcional à percentagem de capacidade de backup utilizada e 400ms. Esse valor mínimo é calculado pela seguinte fórmula: ***percentageStorage * 399 / 100***. Desta forma é menos provável os peers mais cheios guardarem chunks recebidos. Tal como na versão sem enhancements, enquanto esta espera decorre, o peer processa as mensagens ***STORED*** que os outros peers enviam, e guarda no ***storedLog*** os peers que guardam de cada chunk. Depois, quando a espera terminar, o peer verifica no seu ***storedLog*** se o chunk que está a processar tem o grau de replicação percebido maior ou igual ao desejado utilizando a função ***withinDesiredRepDeg(ChunkID chunkID)***. Caso retorne falso, o peer irá guardar o chunk e enviar em multicast a mensagem ***STORED***.

2. Restore

O enhancement do restore protocol é utilizado de forma a evitar a transmissão desnecessária em multicast de chunk (mensagens de grande tamanho) que apenas necessitam de ser entregues a um dos peers.

De forma a evitar o uso de um canal multicast para transferir os chunks, implementámos uma ligação TCP, permitindo assim transferência direta e sem perdas de chunks. O peer iniciador envia, no corpo da mensagem GetChunk, o seu endereço a porta em que está a ouvir. Para o caso de haver vários peers a correr no mesmo PC, utiliza-se a porta peerID % 65536, para evitar ao máximo que sejam utilizadas várias portas ao mesmo tempo. Deste modo, um peer que tenha o chunk liga-se imediatamente ao peer iniciador, sendo que este espera até um máximo de 2 segundos. Qualquer outro peer que tente ligar-se depois disso irá falhar, evitando que sejam enviados vários chunks. O protocolo restore, tal como o de backup, tenta até um máximo de 5 vezes obter o chunk, com intervalos de espera subsequentemente superiores, evitando assim que um pequeno erro (timeout, envio do chunk errado, etc) cause o impedimento da recuperação do ficheiro completo.

3. File Deletion

O enhancement do file deletion protocol tem como objetivo de ao utilizar o protocolo, todas os chunks dos ficheiros armazenados nos outros peers sejam apagados, mesmo que alguns dos peers não estejam a correr no momento, evitando assim que o espaço ocupado pelos chunks nos peers desligados nunca seja recuperado.

De modo a desenvolver o enhancement foram criadas duas mensagens ***CONFIRMDELETION*** e ***START***. Na parte inicial do protocolo o ficheiro a ser apagado pelo peer iniciador vai ser marcado como ***waitingForDeletion*** no seu ***backupLog*** utilizando a função ***addToDeletion(String fileId)***. Quando um peer que contém chunks do ficheiro a ser apagado recebe a mensagem DELETE, responde no MCC com ***CONFIRMDELETION***, após apagar os chunks do ficheiro.

<Version> CONFIRMDELETION <SenderId> <FileId> <CRLF><CRLF>

Um peer, ao receber esta mensagem, vai verificar se o seu ***backupLog*** contém o ficheiro associado a FileId. Caso isso se verifique, executa a função ***removeStorer(String***

fileId,int peerId) no ***backupLog*** que remove o peer SenderId como storer do ficheiro. Quando o grau de replicação percebido do ficheiro se tornar 0, o ficheiro é apagado do ***backupLog***. Um ficheiro que após execução do protocolo delete esteja marcado com waitingForDeletion no ***backupLog*** indica-nos que não foi apagado por todos os peers que continham as suas chunks possivelmente porque não estavam a correr no momento. Para estes casos, quando um peer é iniciado, é enviada para o MCC a mensagem START.

<Version> START <SenderId> ? <CRLF><CRLF>

Ao receber esta mensagem, cada peer vai verificar se o seu ***backupLog*** contém ficheiros marcados com ***waitingForDeletion*** e iniciar o protocolo delete file por cada um deles.

Concorrência

Para tirar proveito máximo do serviço, utilizámos threads, que permitem que vários protocolos sejam executados ao mesmo tempo. Decidimos implementar concorrência até à secção 5.1 das dicas fornecidas no guião: Processamento simultâneo de diferentes mensagens recebidas no mesmo canal, com thread pools. Mais detalhadamente, temos, como base uma thread por cada canal (MCC, MDB, MDR), inicializadas deste modo:

```
new MulticastThread( threadName: "MCCThread", this.peerID, peerState.getMCGroups().MCCGroup, this.peerState).start();
new MulticastThread( threadName: "MDBThread", this.peerID, peerState.getMCGroups().MDBGroup, this.peerState).start();
new MulticastThread( threadName: "MDRThread", this.peerID, peerState.getMCGroups().MDRGroup, this.peerState).start();
```

Esta classe possui uma thread pool de 15 threads. Em loop infinito, lê o canal multicast, analisa a mensagem *(parseMessage)* e, a não ser que não a compreenda ou tenha sido enviada por si próprio, processa-a.

Para o processamento da mensagem, é criada uma worker thread, a partir da thread pool inicialmente obtida, de modo a que o processamento seja feito enquanto outras mensagens sejam recebidas. A thread pool é utilizada para evitar o impacto negativo que o overhead de criação e destruição têm na performance, e também para não haver um número de threads a correr em simultâneo superior ao que o sistema suporta. Para além disso, como o reclaim poderá por si originar um backup, foi utilizada outra Thread Pool apenas para receber as mensagens Stored, pois no caso bastante plausível de estar a processar um grande número de mensagens removed (quando outro peer faz reclaim 0), a Thread Pool não pode ficar sem threads disponíveis para processar as mensagens Stored.

```
private static final ScheduledExecutorService threadPoolStored = Executors.newScheduledThreadPool( corePoolSize: 10);
private static final ScheduledExecutorService threadPoolGeneral = Executors.newScheduledThreadPool( corePoolSize: 15);


public MulticastThread(String threadName, int peerID, MulticastGroup group, PeerState peerState) {
    super(() -> {
        System.out.println("Starting thread: " + threadName);
        while (!Thread.interrupted()) {
            Message message;
            try {
                message = MessageParser.parseMessage(group.receiveFromGroup());
            } catch (IndexOutOfBoundsException | NumberFormatException e) {
                System.out.println(threadName + ": Invalid message syntax. Ignoring... ");
                continue;
            }
            if (peerState.getVersion().equals(Message.VERSION_VANILLA) && message.version.equals(Message.VERSION_ENHANCED)) {
                System.out.println(threadName + ": I am not running version 1.1, Ignoring... ");
                continue;
            }
            if (message.senderId == peerID)
                continue;
            ScheduledExecutorService threadPool = message instanceof StoredMessage ? threadPoolStored : threadPoolGeneral;
            threadPool.execute(new Thread(new MessageWorker(message, peerID, peerState)));
        }
    });
}
```

A worker thread (MessageWorker) simplesmente processa uma mensagem, executando uma função diferente dependendo do tipo.

Para que seja possível o processamento de vários tipos de mensagens ao mesmo tempo tomaram-se os seguintes cuidados ao receber cada tipo de mensagem:

- **Chunk**: Com ConcurrentHashMap<ChunkID, ChunkMessage> (chunkWaitingRoom), guardam-se os ChunkIDs para os quais o peer está à espera de uma mensagem CHUNK. Recebendo uma mensagem CHUNK, o peer simplesmente verifica se o map tem uma entrada com o ChunkID, colocando a mensagem.

```
@Override
public void processChunkMessage(ChunkMessage message) {
    if (chunkWaitingRoom.containsKey(message.chunkID()))
        chunkWaitingRoom.put(message.chunkID(), message);
}
```

- **Delete**: Não foram precisos muitos cuidados extra, já que storedLog já continha um HashMap com os chunks guardados, e o peer simplesmente apaga do Map todas as referências a tal ficheiro. Apenas foi necessário mudar para ConcurrentHashMap.
- **GetChunk**: No caso da versão com enchancements, não é necessário nenhum tipo de cuidados, pois a ligação TCP não precisa de partilhar memória com outras threads. Para a versão 1.0, utiliza-se o ConcurrentHashMap<ChunkID, ChunkMessage> mencionado acima. Ao receber um GETCHUNK para um chunk que o peer guarda, esse ChunkID é adicionado ao HashMap com uma mensagem vazia (NullMessage). Se ao fim dos [0-400]ms não houver nenhuma mensagem não nula para esse ChunkID, significa que nenhum peer enviou a mensagem Chunk durante o intervalo, logo o próprio fá-lo. Independentemente do caso, a entrada no HashMap é removida no final.

```
chunkWaitingRoom.put(message.chunkID(), ChunkMessage.MakeNull());
MulticastThread.sleep(new Random().nextInt( bound: 400));
if (chunkWaitingRoom.get(message.chunkID()).isNull()) {
    Chunk storedChunk = this.storedLog.getChunk(message.chunkID(), this.backupFolderPath);
    Log("Sending chunk to peer " + message.senderId + ": " + storedChunk);
    peerState.getMCGroups().MDRGroup.sendToGroup(new ChunkMessage(message.version, peerID, storedChunk).toString());
}
chunkWaitingRoom.remove(message.chunkID());
```

- **PutChunk**: Para questões de concorrência, o storedLog é bastante importante, pois guarda os peers que enviaram uma mensagem Stored para um certo chunkID. Se ao fim dos [0-400]ms já houver suficiente grau de replicação percebido, a mensagem é ignorada. Também é registado num ConcurrentHashMap<ChunkID, Boolean> (putChunkWaitingRoom) se foi recebido um PutChunk para cada ChunkID. Tal como para a mensagem Chunk, é apenas registado se já houver uma entrada para esse ChunkID, pois significa que estamos à espera dessa mensagem. Esta parte vai ser bastante útil para a mensagem Removed.

```
if (this.putChunkWaitingRoom.containsKey(message.chunkID()))
    this.putChunkWaitingRoom.put(message.chunkID(), true);
```

- **Removed**: Se um chunk que o peer guarda ficar com grau de replicação inferior ao desejado, chunkID é adicionado ao HashMap descrito acima, com valor false. Assim é fácil verificar se, passados [0-400]ms, o peer recebeu um PutChunk.

```
this.putChunkWaitingRoom.put(message.chunkID(), false);

MulticastThread.sleep(new Random().nextInt( bound: 400));

if (!this.putChunkWaitingRoom.get(message.chunkID())) {
    Log("Chunk fell below desired replication degree! Initiating backup of " + message.chunkID());
    try {
        Chunk chunkToBackup = this.storedLog.getChunk(message.chunkID(), this.backupFolderPath);
        new ChunkBackup(chunkToBackup, peerID, peerState).backup(this.storedLog.getDesiredRepDeg(message.chunkID()), this.storedLog);
        peerState.getMCGroups().MCCGroup.sendToGroup(new StoredMessage(message.version, this.peerID, chunkToBackup).toString());
    } catch (Exception e) {
        Log("Error backing up chunk. Error message: " + e.getMessage());
    }
}
this.putChunkWaitingRoom.remove(message.chunkID());
```

- **Stored**: Bastante fácil de implementar com storedLog e backupLog. Essencialmente, verifica se possui em algum dos hashmaps uma entrada para o chunkID. Nesse caso, adiciona o peerID como storer.

```
@Override
public void processStoredMessage(StoredMessage message) {
    if (storedLog.hasChunk(message.chunkID()))
        storedLog.addStorer(message.chunkID(), message.senderId);
    if (backupLog.hasFile(message.fileId))
        backupLog.addStorer(message.chunkID(), message.senderId);
}
```