

SDIS 2021

Distributed Backup Service

Relatório - 1º Trabalho laboratorial

Diana Freitas, [up201806230](#)

Eduardo Brito, [up201806271](#)

Turma 7, Grupo 4

1. Protocolo Backup

Na versão 1.0 deste protocolo, a estratégia usada era independente do Replication Degree especificado e levava a que todos os peers com espaço disponível armazenassem chunks do ficheiro em questão, sempre que recebessem uma mensagem PUTCHUNK. Como assinalado no enunciado, esta abordagem esgota muito rapidamente o espaço útil de cada peer e causa muita atividade nos canais envolvidos, além de se revelar desnecessário um Replication Degree maior que o desejado.

A versão 2.0 vem resolver este problema, forçando à otimização do uso do espaço de armazenamento de cada peer. Na nossa implementação, cada peer escuta as mensagens STORED dos restantes, guardando o número de confirmações de cada CHUNK. Assim, antes de o armazenar, verifica se, para aquele CHUNK, o Replication Degree desejado já foi, de facto, atingido, isto é, verifica se o número de confirmações enviadas por outros peers corresponde ao Replication Degree, evitando gastar espaço para o guardar.

Storage.java

```
...
// To store the chunks confirmations
private ConcurrentHashMap<ChunkKey, HashSet<Integer>> confirmedChunks;
...
/**
 * Used by any peer to store all the STORED confirmations received for chunks,
 * except his own confirmations.
 * @param chunkKey
 * @param serverId
 */
public void addStoredConfirmation(ChunkKey chunkKey, int serverId) {
    if (serverId != Utils.PEER_ID) {
        HashSet<Integer> peers;
        if (!this.confirmedChunks.containsKey(chunkKey)) {
            peers = new HashSet<>();
        } else {
            peers = this.confirmedChunks.get(chunkKey);
        }
        peers.add(serverId);
        this.confirmedChunks.put(chunkKey, peers);
    }
}
```

A função é chamada na classe **ControlReceiver**, sempre que é recebida uma mensagem STORED, guardando a confirmação do peer que enviou a mensagem relativa ao chunk recebido. Esta informação é armazenada no HashSet do chunk respetivo, permitindo ignorar confirmações repetidas.

PutChunkHandler.java

```
...
// In v2.0, if the chunk has the desired replication degree, abort the operation
if (this.message.getVersion().equals("2.0")
    &&
    (storage.getConfirmedChunks(chunk.getChunkKey()) >= chunk.getReplicationDegree())) {
    return;
}
```

Antes de guardar/confirmar um CHUNK, é verificado se o seu replication degree já foi atingido.

2. Protocolo Restore

Na versão 1.0 do protocolo, quando o initiator-peer envia uma mensagem GETCHUNK, as respostas vão ser enviadas através do canal MDR, o que leva a que todos os peers recebam as mensagens, quando, na verdade, só o initiator-peer precisa de as receber.

Para resolver esta questão, na versão 2.0, é utilizado o TCP para enviar as mensagens CHUNK de resposta ao pedido de Restore. Isto permite enviar diretamente as respostas ao peer que, de facto, fez o pedido. Para garantir que peers de versões diferentes podem interoperar, o MDR é mantido e utilizado pelos peers da versão 1.0. Já os peers da versão 2.0 que recebem um pedido de Restore, verificam a versão da mensagem recebida para saberem se devem utilizar o MDR ou o socket TCP para enviar a resposta. Usando o protocolo TCP, as mensagens GETCHUNK têm um campo adicional que corresponde à porta onde o initiator-peer está a aguardar a resposta.

TCPRestoreReceiver.java

```
/**
 * Accept new TCP connections and start a thread for each one of them
 */
@Override
public void run() {
    while (true) {
        try {
            this.peer.getScheduler().execute(new
                TcpRestoreHandler(this.peer, this, serverSocket.accept()));
        } catch (IOException e) {
            Utils.protocolError(Protocol.RESTORE, null,
                " I/O error when waiting for a connection in TCP socket");
        }
    }
}
```

Na versão 2.0, para além de serem lidas as mensagens do MDR, é utilizada a classe **TCPRestoreReceiver** para receber as respostas ao pedido de restore. Para cada mensagem recebida, é lançada uma thread, permitindo assim o processamento simultâneo das respostas.

GetChunkHandler.java

```
// Version 2.0: use TCP to send CHUNK
if (Utils.PROTOCOL_VERSION.equals("2.0") &&
    (this.address != null) &&
    (this.tcpPort >= 0)) {
    Socket socket = new Socket(this.address, this.tcpPort);
    DataOutputStream out = new DataOutputStream(socket.getOutputStream());
    out.write(packet.getData());
    out.close();
    socket.close();
}
// Version 1.0: use Restore Multicast Channel to send CHUNK
else {
    restoreChannel.sendMessage(packet);
}
```

A classe **GetChunkHandler** é responsável por garantir a interoperabilidade entre os peers, conduzindo a resposta pelo canal ideal. Se o pedido GETCHUNK e o peer que o recebe tiverem ambos a versão 2.0, a resposta é enviada por TCP, senão é enviada para o canal MultiCast.

3. Protocolo Delete

Na versão 1.0, como as mensagens DELETE não requerem nenhuma resposta, não há maneira de confirmar a remoção do ficheiro e respetivos chunks do sistema. Além disso, se um dos peers não estiver ligado no momento do pedido DELETE, quando se conectar novamente ao sistema, acaba por não eliminar o conteúdo suposto.

A versão 2.0 permite resolver o problema anterior, adicionando dois novos tipos de mensagem ao protocolo. O primeiro tipo corresponde à resposta efetiva ao pedido DELETE, através de uma confirmação DELETED enviada no MC, que é armazenada em memória não volátil pelo peer que fez o pedido inicial. Estas confirmações são contabilizadas para garantir que todos os peers apagam o conteúdo. O segundo tipo de mensagem, GETDELETE, é enviado sempre que um peer se liga ao serviço e permite que os peers que não estavam ligados ao sistema durante algum pedido DELETE possam ficar a par das mudanças e efetuar a eliminação dos chunks. Assim, ao receber um GETDELETE, um peer que não tenha recebido confirmações DELETED de todos os peers que tinham guardado chunks de um ficheiro eliminado, verifica se está a aguardar alguma confirmação por parte do peer que enviou o GETDELETE e, em caso afirmativo, inicia o protocolo DELETE para esse ficheiro.

Storage.java

```
...
// For each deleted file, keeps the peers that did not confirm the deletion
private ConcurrentHashMap<String, HashSet<Integer>> deletedFiles;
...
/**
 * In the version 2.0 of the Delete Protocol, it is used by the Initiator-peer to add a file
 * that is being deleted and associates it with the peers that must send DELETED
 * confirmations.
 */
public void addDeletedFile(String fileId){
    HashSet<Integer> peers = new HashSet<>();
    for (ChunkKey key : this.confirmedChunks.keySet()) {
        if(key.getFileId().equals(fileId)){
            peers.addAll(this.confirmedChunks.get(key));
        }
    }
    this.deletedFiles.put(fileId, peers);
}
```

Ao iniciar um protocolo DELETE, esta função é utilizada para associar um ficheiro a ser eliminado com os peers que guardaram algum chunk desse mesmo ficheiro. Assim, ao receber uma confirmação DELETED, o initiator-peer apenas precisa de remover do HashSet associado ao ficheiro nos **deletedFiles** o id do peer que confirmou a sua eliminação.

GetDeleteHandler.java

```
ConcurrentHashMap<String, HashSet<Integer>> deletedFiles = storage.getDeletedFiles();
ControlChannel controlChannel = this.peer.getControlChannel();
for(String file : deletedFiles.keySet()){
    try {
        if(deletedFiles.get(file).contains(this.senderId)){
            DatagramPacket packet =
                controlChannel.deletePacket(Utils.PROTOCOL_VERSION,Utils.PEER_ID,file);
            controlChannel.sendMessage(packet);
        }
    } ...
}
```

Ao receber uma mensagem GETDELETE, um peer lança uma thread da classe **GetDeleteHandler** onde verifica se está a aguardar a confirmação DELETED do peer que enviou a mensagem e, em caso afirmativo, inicia o protocolo DELETE.

4. Concorrência

De forma a suportar a execução concorrente dos protocolos e escalar a implementação do projeto, todas as sugestões propostas foram implementadas ao detalhe. Em primeiro lugar, na nossa implementação é usada uma Thread por cada Multicast Channel, para receber, ao mesmo tempo, mensagens de diferentes protocolos. Estas Threads comportam-se como Receivers e, a cada mensagem recebida, delegam o processamento para Worker Threads que trabalham concorrentemente. As Threads são criadas usando um objeto da classe **ScheduledThreadPoolExecutor**, que permite não só evitar a criação sucessiva de novas Threads, reutilizando as existentes, como também permite lidar com os Timeouts de uma forma mais efetiva, evitando os problemas de escalabilidade que seriam causados pelo uso de mecanismos como **Thread.sleep()**.

Para suportar o processamento concorrente de dados do sistema, com a execução simultânea de várias Threads, as nossas estruturas de dados são do tipo **ConcurrentHashMap**. Estas estruturas suportam conjuntos de operações em sequência, assim como em paralelo e com grande volume, sendo desenhadas para serem seguras quando usadas concorrentemente por diferentes Threads.

Como última medida para promover a escalabilidade, usámos objetos da classe **AsynchronousFileChannel**, eliminando as *blocking-calls* que restavam, isto é, as chamadas ao sistema de ficheiros, para escrita e leitura de chunks e das informações constantes no espaço de armazenamento dos diferentes peers.

4.1. Receiver e Worker Threads

ChannelAggregator.java

```
public void run(Peer peer) {
    this.backupChannel.run(peer);
    this.restoreChannel.run(peer);
    this.controlChannel.run(peer);
}
```

Na classe **ChannelAggregator** é chamado o método **run** de cada Multicast Channel, que, por sua vez, inicia uma thread que estende a classe **MessageReceiver** e que se responsabiliza pela leitura das mensagens recebidas nesse canal, delegando o seu processamento para Worker Threads.

BackupChannel.java

```
@Override
public void run(Peer peer) {
    this.messageReceiver = new BackupReceiver(peer);
    peer.getScheduler().execute(this.messageReceiver);
}
```

Como exemplo, no método **run** do **BackupChannel**, é criada uma instância da classe **BackupReceiver**, responsável por receber mensagens do tipo PUTCHUNK.

BackupReceiver.java

```
// Wait a random delay before saving the chunk and sending the STORED confirmation
this.peer.getScheduler().schedule(
    new PutChunkHandler(this.peer, message),
    Utils.getRandomDelay(),
    TimeUnit.MILLISECONDS);
```

Como exemplo, no **BackupReceiver**, ao ser recebida uma mensagem do tipo PUTCHUNK, a responsabilidade de armazenar o CHUNK recebido é delegada a uma nova Thread do tipo **PutChunkHandler**.

4.2. ConcurrentHashMap

Storage.java

```
...
// Stored Chunks
private ConcurrentHashMap<ChunkKey, Integer> storedChunks;
// To store the chunks confirmations
private ConcurrentHashMap<ChunkKey, HashSet<Integer>> confirmedChunks;
// To retrieve information about files which the peer has initiated a backup for
(initiator)
private ConcurrentHashMap<String, SFile> backupFiles;
// For each deleted file, keeps the peers that did not confirm the deletion
private ConcurrentHashMap<String, HashSet<Integer>> deletedFiles;
...
```

Peer.java

```
...
// For each file pending restore, keeps the chunks already restored (initiator-peer)
private ConcurrentHashMap<String, HashSet<Chunk>> pendingRestoreFiles;
// Keeps track of restore requests (non-initiator peers)
private ConcurrentHashMap<ChunkKey, Integer> restoreRequests;

// Auxiliar data structures for REMOVED
// Keeps track of removed chunks
private ConcurrentHashMap<ChunkKey, Integer> removedChunks;
...
```

4.3. AsynchronousFileChannel

Storage.java

```
public void store(Chunk chunk, ExecutorService scheduler) throws IOException {
    String fileDir = this.path + "/backup/file-" + chunk.getFileId();
    Files.createDirectories(Paths.get(fileDir));
    Path path = Paths.get(fileDir + "/chunk-" + chunk.getChunkNum() + ".ser");
    this.capacityUsed += chunk.getBuffer().length;

    Set<OpenOption> options = new HashSet<OpenOption>();
    options.add(StandardOpenOption.CREATE);
    options.add(StandardOpenOption.WRITE);

    AsynchronousFileChannel channel = AsynchronousFileChannel.open(path, options,
scheduler);
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    ObjectOutputStream oos = new ObjectOutputStream(baos);
    oos.writeObject(chunk);
    oos.flush();

    ByteBuffer buffer = ByteBuffer.wrap(baos.toByteArray());
    Future<Integer> operation = channel.write(buffer, 0);
    while (!operation.isDone()) {}
}
```

```

        channel.close();
        oos.close();
        baos.close();
    }

    public Chunk read(String fileId, int chunkNum, ExecutorService scheduler) throws
IOException,      ClassNotFoundException {
        Path path = Paths.get(this.path+"/backup/file-"+fileId+"/chunk-"+chunkNum+".ser");

        Set<OpenOption> options = new HashSet<OpenOption>();
        options.add(StandardOpenOption.READ);

        AsynchronousFileChannel channel = AsynchronousFileChannel.open(path, options,
scheduler);

        ByteBuffer buffer = ByteBuffer.allocate(Utils.CHUNK_SIZE * 2);
        Future<Integer> result = channel.read(buffer, 0);
        while (!result.isDone()) {}
        buffer.flip();

        ByteArrayInputStream bais = new ByteArrayInputStream(buffer.array());
        ObjectInputStream ois = new ObjectInputStream(bais);
        Chunk c = (Chunk) ois.readObject();
        return c;
    }
}

```

O método **store** recorre a uma instância da classe **AsynchronousFileChannel** para armazenar e ler os CHUNKS serializados em memória não volátil, evitando *blocking-calls* ao sistema de ficheiros.