U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

12 de abril de 2021

Relatório Projeto 1

SISTEMAS DISTRIBUÍDOS

Grupo T1G06

Beatriz Mendes up201806551
José Rodrigues up201708806

Introdução

Este relatório tem como foco explicar detalhadamente o *design* implementado no projeto 1 da unidade curricular Sistemas Distribuídos (*Distributed Backup Service*) pelo nosso grupo de trabalho que permite a execução simultânea de subprotocolos.

Execução Simultânea de Sub-Protocolos (*Concurrency*)

Relativamente à execução simultânea de subprotocolos, tivemos como base a documentação dada pelo professor Pedro Souto (*Hints for Concurrency in Project 1* - https://web.fe.up.pt/~pfs/aulas/sd2021/projs/proj1/concurrency_hints.html).

Numa primeira abordagem, na criação dos canais de comunicação, utilizamos a classe *Thread*. Assim, é possível transmitir mensagens entre diferentes canais sem haver colisão ou utilização destes em simultâneo, sendo a maneira mais segura de passar informação entre os *Peers*. Note-se de seguida o exemplo mencionado anteriormente.

```
public synchronized static void openChannels() throws UnknownHostException {
    mc = new Channel(peer_id, mc_maddress, mc_port, ChannelType.MC);
    mdb = new Channel(peer_id, mdb_maddress, mdb_port, ChannelType.MDB);
    mdr = new Channel(peer_id, mdr_maddress, mdr_port, ChannelType.MDR);

    mc_thread = new Thread(mc);
    mdb_thread = new Thread(mdb);
    mdr_thread = new Thread(mdr);

    mc_thread.start();
    mdb_thread.start();
    mdr_thread.start();
}
```

De seguida, de forma a garantir o sucesso na execução simultânea de protocolos, notamos que deveríamos alterar as *HashMaps* que possuíamos ao longo do projeto para a estrutura de dados *ConcurrentHashMaps* (Secção 4). Esta estrutura é a mais adequada uma vez fornece alta simultaneidade em ambientes com múltiplas *threads* e a operação de escrita é feita com um *lock*, impedindo assim a escrita em simultâneo por várias *threads*. Utilizamos esta estrutura de dados quando queremos guardar o *replication degree* atual de um determinado *chunk*. Assim, cria-se uma chave com o *fileID* e o *chunkNo* do *chunk* em questão e associa-se a esta o valor atual do *replication degree*, que pode ser alterado em múltiplas *threads*.

```
private ConcurrentHashMap<String, Integer> occurrences;
```

Posteriormente, outra melhoria implementada pelo nosso grupo de trabalho está relacionada com o uso de *ScheduledThreadPoolExecutor*'s (Secção 5 e 6). Esta classe evita a criação de novas *threads* para cada mensagem e ainda permite a criação de um gestor de tempo limite. Este gestor permite que a ação que pretendemos realizar aconteça apenas ao fim do *delay* determinado. No exemplo do *channelExecuters*, é verificável a utilização de um *ScheduledThreadPoolExecutor* sem um *delay* determinado (*execute()*), enquanto que no exemplo do *peerExecutors* esta mesma classe tem um *delay* igual a *response_time* (*schedule()*).

```
if(!parts[2].equals(String.valueOf(peer_id))) {
    channelExecuters.execute(() -> {
        System.out.println("> Peer " + Peer.getPeerID() + ": Catched message on channel " + channelType.toString() + " from peer " + parts[2]);
        Peer.treatMessage(p);
    });
}
```

```
peerExecutors.schedule(() -> {
    // <Version> STORED <SenderId> <FileId> <ChunkNo> <CRLF><CRLF>
    String response = version + " STORED " + peer_id + " " + file_id + " " + chunk_no + "\r\n\r\n";
    mc.sendMessage(response.getBytes());
}, response_time, TimeUnit.MILLISECONDS);
```

Além destas implementações, o nosso grupo também tirou partido da sincronização em Java que, segundo a documentação, é a capacidade de controlar o acesso a múltiplas *threads* a qualquer recurso partilhado de forma segura. Para tal, apenas foi necessário adicionar a *keyword* "*synchronized*" aos métodos que iriam necessitar desta propriedade.

```
public static synchronized void treatMessage(DatagramPacket packet) {
```

Por fim, de forma a remover todos os bloqueios que possam surgir na criação, escrita e leitura de ficheiros, utilizamos a classe *AsynchronousFileChannel*. Esta implementação verifica-se, por exemplo, na criação de *chunks*.

```
/**
 * Function used to perform the backup of a chunk
 */
public synchronized void performBackup() {
    try {
        Path path = Path.of("peer " + pID + "\\" + "chunks\\" + fileId + "_" + chunkNo);
        AsynchronousFileChannel fileChannel = AsynchronousFileChannel.open(path, StandardOpenOption.CREATE, StandardOpenOption.WRITE);
        fileChannel.write(ByteBuffer.wrap(content), 0);
        fileChannel.close();
        System.out.println("> Peer " + pID + ": saved chunk nº" + chunkNo + " of file with fileID: " + fileId);
    } catch (IOException e) {
        System.err.println("> Peer " + pID + " exception: failed to save chunk " +  chunkNo);
        e.printStackTrace();
    }
}
```

Conclusão

Em suma, o nosso grupo de trabalho, após múltiplos testes, considera que o serviço implementado por nós permite a execução simultânea de subprotocolos graças às dicas para presentes na documentação fornecida ao longo das aulas da unidade curricular.