Faculdade de Engenharia da Universidade Do Porto

Mestrado Integrado em Engenharia Informática e Computação

Sistemas Distribuídos – 3º ano

TP1 – Distributed Backup Service

Alunos (T3G11):

João Miguel Vaz Tello da Gama Amaral – up201708805

João Nuno Rodrigues Ferreira – up201605330

14 de Abril, 2019

Introdução

O seguinte relatório tem como objetivo explicar e descrever detalhadamente o design escolhido que permite a execução simultânea de protocolos no protocolo base do projeto: Distributed backup service.

O projeto foi desenvolvido no âmbito da unidade curricular de Sistemas Distribuídos (SDIS).

Execução simultânea de protocolos

Relativamente ao design que foi implementado para permitir a execução simultânea de protocolos, o grupo teve em conta vários fatores.

Threads:

A classe **Peer** tem um atributo por canal multicast: **MC**, **MDB** e **MDR**. Para que exista apenas uma thread por canal multicast, no método **main** do **Peer** é executada uma thread para cada um dos canais.

```
public static void main(String[] args){

    try{

        if(args.length != 9){
            System.out.println("Usage:\n");
            System.out.println("java Peer <serverID> <protocolVersion> <serviceAccessPoint> <ipAddressMC> <portMC> <ipAddressMDB> <portMDB> <ipAddressMDR> <portMDR>");
            return;
        }

        serverID = "P" + args[0];
        protocolVersion = Double.parseDouble(args[1]);
        String serviceAccessPoint = args[2];
        String ipAddressMC = args[3];
        int portMC = Integer.parseInt(args[4]);
        String ipAddressMDB = args[5];
        int portMDB = Integer.parseInt(args[6]);
        String ipAddressMDR = args[7];
        int portMDR = Integer.parseInt(args[8]);

        Peer peer = new Peer(ipAddressMC, portMC, ipAddressMDB, portMDB, ipAddressMDR, portMDR);

        RMISystem rmiSystem = (RMISystem) UnicastRemoteObject.exportObject(peer, 0);
        Registry registry = LocateRegistry.getRegistry();
        registry.bind(serverID, rmiSystem);

    } catch (Exception exception) {

        exception.printStackTrace();
    }

    new Thread(MC).start();
    new Thread(MDB).start();
    new Thread(MDR).start();
}
```

Quando o programa é inicializado, através da classe **TestApp**, é possível identificar que protocolo utilizar. Após identificação do **BACKUP**, **RESTORE**, **DELETE**, **RECLAIM** ou **STATE** é criada uma thread para o protocolo escolhido para que seja possível serem executados vários ao mesmo tempo (todos são thread menos o **STATE**).

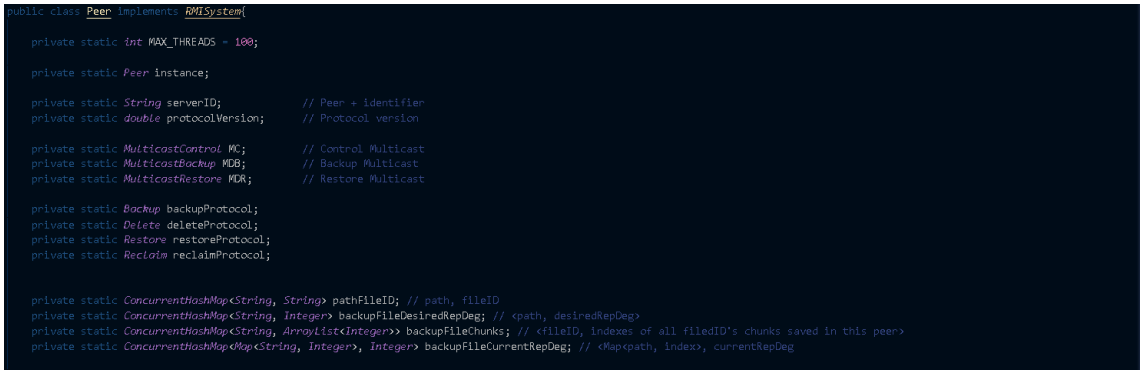
```
public void backupData(String path, int repDeg){

    Backup backupProtocol = new Backup(path, repDeg);
    new Thread(backupProtocol).start();
}

public void deleteData(String path){

    deleteProtocol = new Delete(path);
    new Thread(deleteProtocol).start();
}

public void restoreData(String path){

    restoreProtocol = new Restore(path);
    new Thread(restoreProtocol).start();
}

public void reclaimSpace(int wantedSpace){

    reclaimProtocol = new Reclaim(wantedSpace);
    new Thread(reclaimProtocol).start();
}
```

Cada canal dá extend á classe **Multicast** onde sempre que é recebida uma mensagem, esta cria uma classe **MessageManager** que trata do seu processamento. Visto que cada canal é uma thread, é possível processar várias mensagens ao mesmo tempo.

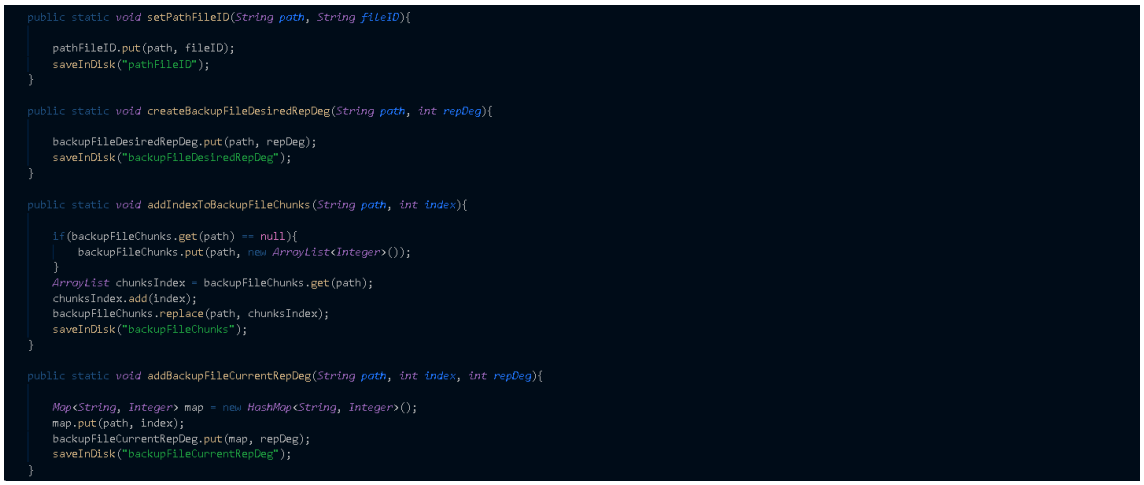
ConcurrentHashMap:

No que toca a estruturas de dados, no caso das tabelas, foi decidido utilizar a estrutura ConcurrentHashMap em vez de HashMap. A estrutura utilizada é adequada para ambientes multi-thread por ser mais segura, ter um melhor desempenho e escalável. A estrutura ConcurrentHashMap tem também um ótimo desempenho quando o número de threads de leitura é maior que o número de escrita.

São utilizados quatro ConcurrentHashMap para guardar dados necessários para os protocolos. São guardados valores como path, fileID, desiredRepDeg e chunks.



```
public class Peer implements RMISystem{

    private static int MAX_THREADS = 100;

    private static Peer instance;

    private static String serverID;             // Peer + identifier
    private static double protocolVersion;      // Protocol version

    private static MulticostControl MC;         // Control Multicast
    private static MulticostBackup MDB;         // Backup Multicast
    private static MulticostRestore MDR;        // Restore Multicast

    private static Backup backupProtocol;
    private static Delete deleteProtocol;
    private static Restore restoreProtocol;
    private static Reclaim reclaimProtocol;


    private static ConcurrentHashMap<String, String> pathFileID; // path, fileID
    private static ConcurrentHashMap<String, Integer> backupFileDesiredRepDeg; // <path, desiredRepDeg>
    private static ConcurrentHashMap<String, ArrayList<Integer>> backupFileChunks; // <fileID, indexes of all filedID's chunks saved in this peer>
    private static ConcurrentHashMap<Map<String, Integer>, Integer> backupFileCurrentRepDeg; // <Map<path, index>, currentRepDeg
```



```
public static void setPathFileID(String path, String fileID){

    pathFileID.put(path, fileID);
    saveInDisk("pathFileID");
}

public static void createBackupFileDesiredRepDeg(String path, int repDeg){

    backupFileDesiredRepDeg.put(path, repDeg);
    saveInDisk("backupFileDesiredRepDeg");
}

public static void addIndexToBackupFileChunks(String path, int index){

    if(backupFileChunks.get(path) == null){
        backupFileChunks.put(path, new ArrayList<Integer>());
    }
    ArrayList chunksIndex = backupFileChunks.get(path);
    chunksIndex.add(index);
    backupFileChunks.replace(path, chunksIndex);
    saveInDisk("backupFileChunks");
}

public static void addBackupFileCurrentRepDeg(String path, int index, int repDeg){

    Map<String, Integer> map = new HashMap<String, Integer>();
    map.put(path, index);
    backupFileCurrentRepDeg.put(map, repDeg);
    saveInDisk("backupFileCurrentRepDeg");
}
```