2020

SDIS Projeto 1

BACKUP SERVICE

ANA MACEDO UP201909572; MIGUEL CASTRO UP201909578

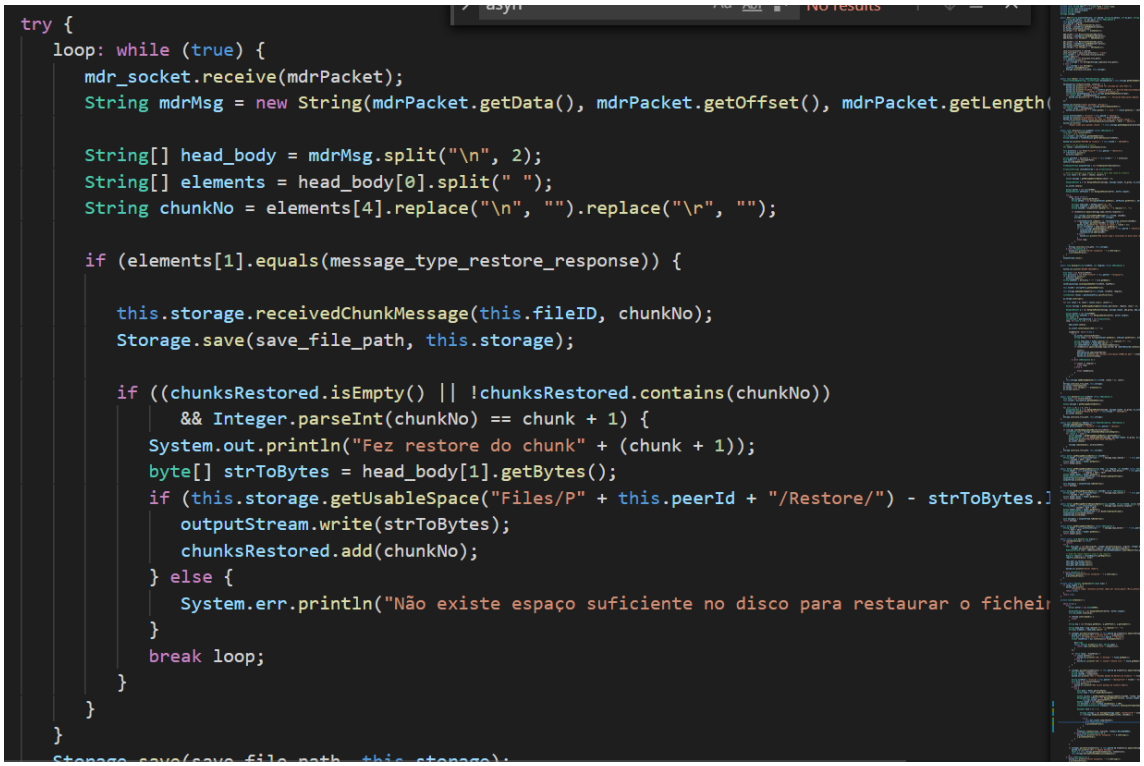
Design de Concorrência

Ao mesmo tempo que se inicializam os *Peers*, no seu construtor, também se inicializam os canais MC, MDB e MDR, subscrevemos os grupos (com a porta e endereço dados como argumento na invocação do *peer)* e criamos as *threads,* uma por canal.

```
public Peer(String protocolVersion, int peerId, String mc_adress, int mc_port, String mdb_adress, int r
    String mdr_adress, int mdr_port) throws IOException {
  this.protocolVersion = protocolVersion;
  this.peerId = peerId;
  mc_socket = new MulticastSocket(mc_port);
  mc_group = InetAddress.getByName(mc_adress);
  mc_socket.joinGroup(mc_group);
  mc_thread = new Thread(() -> mcChannel());

  mdb_socket = new MulticastSocket(mdb_port);
  mdb_group = InetAddress.getByName(mdb_adress);
  mdb_thread = new Thread(() -> mdbChannel());

  mdr_socket = new MulticastSocket(mdr_port);
  mdr_group = InetAddress.getByName(mdr_adress);
  mdr_socket.joinGroup(mdr_group);
  mdr_thread = new Thread(() -> mdrChannel());

  save_file_directory += peerId;
  save_file_path = save_file_directory + "/save";
```

One Thread per Multicast Channel, Many Protocol Instances at a Time

Usamos uma thread receptora por canal, e cada uma completa o processamento de uma mensagem, antes de iniciar o processamento da seguinte mensagem, como se vê na figura seguinte que representa a receção da mensagem CHUNK no protocolo Restore e o tratamento do corpo da mensagem recebido.



```
try {
  loop: while (true) {
    mdr_socket.receive(mdrPacket);
    String mdrMsg = new String(mdrPacket.getData(), mdrPacket.getOffset(), mdrPacket.getLength(

    String[] head_body = mdrMsg.split("\n", 2);
    String[] elements = head_body[0].split(" ");
    String chunkNo = elements[4].replace("\n", "").replace("\r", "");

    if (elements[1].equals(message_type_restore_response)) {

      this.storage.receivedChunkMessage(this.fileID, chunkNo);
      Storage.save(save_file_path, this.storage);

      if ((chunksRestored.isEmpty() || !chunksRestored.contains(chunkNo))
          && Integer.parseInt(chunkNo) == chunk + 1) {
        System.out.println("Fez restore do chunk" + (chunk + 1));
        byte[] strToBytes = head_body[1].getBytes();
        if (this.storage.getUsableSpace("Files/P" + this.peerId + "/Restore/") - strToBytes.l
          outputStream.write(strToBytes);
          chunksRestored.add(chunkNo);
        } else {
          System.err.println("Não existe espaço suficiente no disco para restaurar o fichei
        }
        break loop;
      }
    }
  }
```

Assim, cada thread executa um loop infinito e em cada iteração recebe a mensagem no respetivo canal multicast e processa a mensagem recebida.

Uso de listas/*Maps* sincronizados (thread-safe)

Na classe Storage várias threads podem aceder aos seus objetos e alterá-los ao mesmo tempo, não sendo o desejado. Para solucionar este problema é necessário o uso da classe thread safe, como a ConcurrentHashMaps e Collections.synchronizedList.

Na figura seguinte encontra-se inicializações de objetos das classes referidas.

```
public Storage(Storage s) {
    this.filesBackedUp = new ConcurrentHashMap<>(s.getBackedUpFiles());
    this.chunksBackedUp = Collections.synchronizedList(new ArrayList<>(s.getChunksBackedUp()));
    this.storedChunks = Collections.synchronizedList(new ArrayList<>(s.getStoredChunks()));
    this.maxDiskSpace = s.getMaxDiskSpace();
    receivedChunkRestore = Collections.synchronizedList(new ArrayList<>());
```

Para além disso, os métodos onde se alteram esses objetos são declarados como *synchronized.*

```
    public synchronized void addBackedUpChunk(String fileId, int chunkNo, int repDegree) {
        Chunk chunk = new Chunk(fileId, chunkNo, repDegree);
        if (!chunksBackedUp.contains(chunk))
            chunksBackedUp.add(chunk);
    }
```

Implementação escalável

Para vários subprotocolos, era necessário cada *Peer* esperar um tempo aleatório uniformemente distribuído entre 0 e 400 ms antes de enviar uma mensagem para o *initiator Peer* ou para executar o subprotocolo backup.

Para uma implementação escalável foi substituída a função *Thread.sleep()* pela classe *java.util.concurrent.ScheduledExecutorService.*

Na imagem encontra-se um exemplo de implementação, em que o *mseconds* são os milissegundos gerados aleatoriamente.

```
    Random random = new Random();
    int mseconds = (int) (random.nextDouble() * 400);
    ScheduledExecutorService scheduler = Executors.newSingleThreadScheduledExecutor();

    Runnable task = () -> {

       Storage storage = new Storage(Storage.load("./SaveFiles/P" + elements[2] + "/save"));
       if (!storage.hasReceivedChunkMessgage(fileId, chunkNo)) {

          try {
             this.mdr_socket.send(sPacket);
          } catch (IOException e) {
             e.printStackTrace();
          }
       }
    };
    scheduler.schedule(task, mseconds, TimeUnit.MILLISECONDS);
```

Neste exemplo, no subprotocolo restore, só se envia a mensagem CHUNK depois de um tempo aleatório.

A função Schedule permite que o que está dentro do bloco *task* seja executado depois dos milissegundos dados.

Este método é repetido no subprotocolo backup e reclaim, sendo as task() diferentes.

Em relação à classe java.nio.channels.AsynchronousFileChannel foi tentada implementar, mas sem sucesso.

Comunicação entre o TestApp e o Initiator Peer

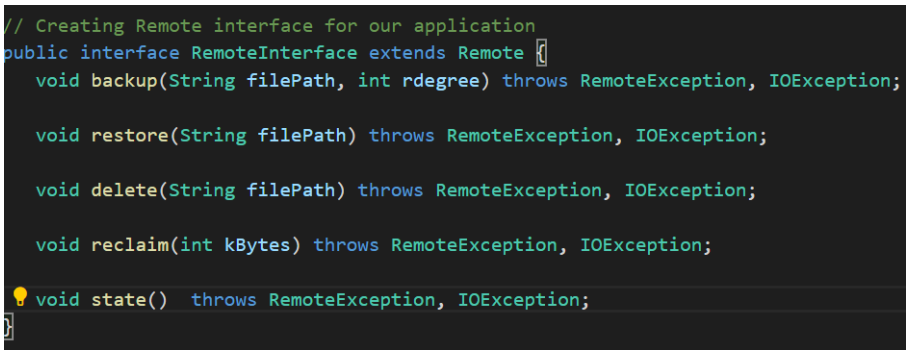
Para a comunicação entre o cliente e o *initiator peer* foi usado RMI.

O nome da referência remota usado no método lookUp() do cliente e no bind() *do initiator peer* é o *peer access point* dado no primeiro argumento, quando se invoca o cliente.

Nas figuras a baixo encontram-se blocos de código da implementação do RMI, TestApp, Peer e RemoteInterface respetivamente.

```
// Getting the registry
Registry registry = LocateRegistry.getRegistry(null);

// Looking up the registry for the remote object
RemoteInterface stub = (RemoteInterface) registry.lookup(args[0]);

if (args[1].equals("BACKUP"))
   stub.backup(args[2], Integer.parseInt(args[3]));
else if (args[1].equals("RESTORE"))
   stub.restore(args[2]);
else if (args[1].equals("DELETE"))
   stub.delete(args[2]);
else if (args[1].equals("RESTORE"))
   stub.reclaim(Integer.parseInt(args[2]));
else if (args[1].equals("STATE")){
   stub.state();
}
```

```
Run | Debug
public static void main(String args[]) {
   if (validacoes(args) == false)
      return;
   try {
      Peer this_peer = new Peer(args[0], Integer.parseInt(args[1]), args[3], Integer.parseInt(args[4]),
            Integer.parseInt(args[6]), args[7], Integer.parseInt(args[8]));
      RemoteInterface stub = (RemoteInterface) UnicastRemoteObject.exportObject(this_peer, 0);

      // Bind the remote object's stub in the registry
      Registry registry = LocateRegistry.getRegistry();
      registry.bind(args[2], stub);
```



```java
// Creating Remote interface for our application
public interface RemoteInterface extends Remote {
   void backup(String filePath, int rdegree) throws RemoteException, IOException;

   void restore(String filePath) throws RemoteException, IOException;

   void delete(String filePath) throws RemoteException, IOException;

   void reclaim(int kBytes) throws RemoteException, IOException;

   void state()  throws RemoteException, IOException;
}
```

Melhorias

Não foram implementadas nenhumas melhorias no projeto.