Instituto Politécnico de Viseu
Escola Superior de Tecnologia e Gestão de Viseu
Departamento de Informática

Unidade Curricular: Sistemas Distribuídos

Relatório Relativo ao Trabalho Final

Realizado por:

Carlos Daniel Ferreira Silva – 20255

Pedro Guedes Monteiro – 20272

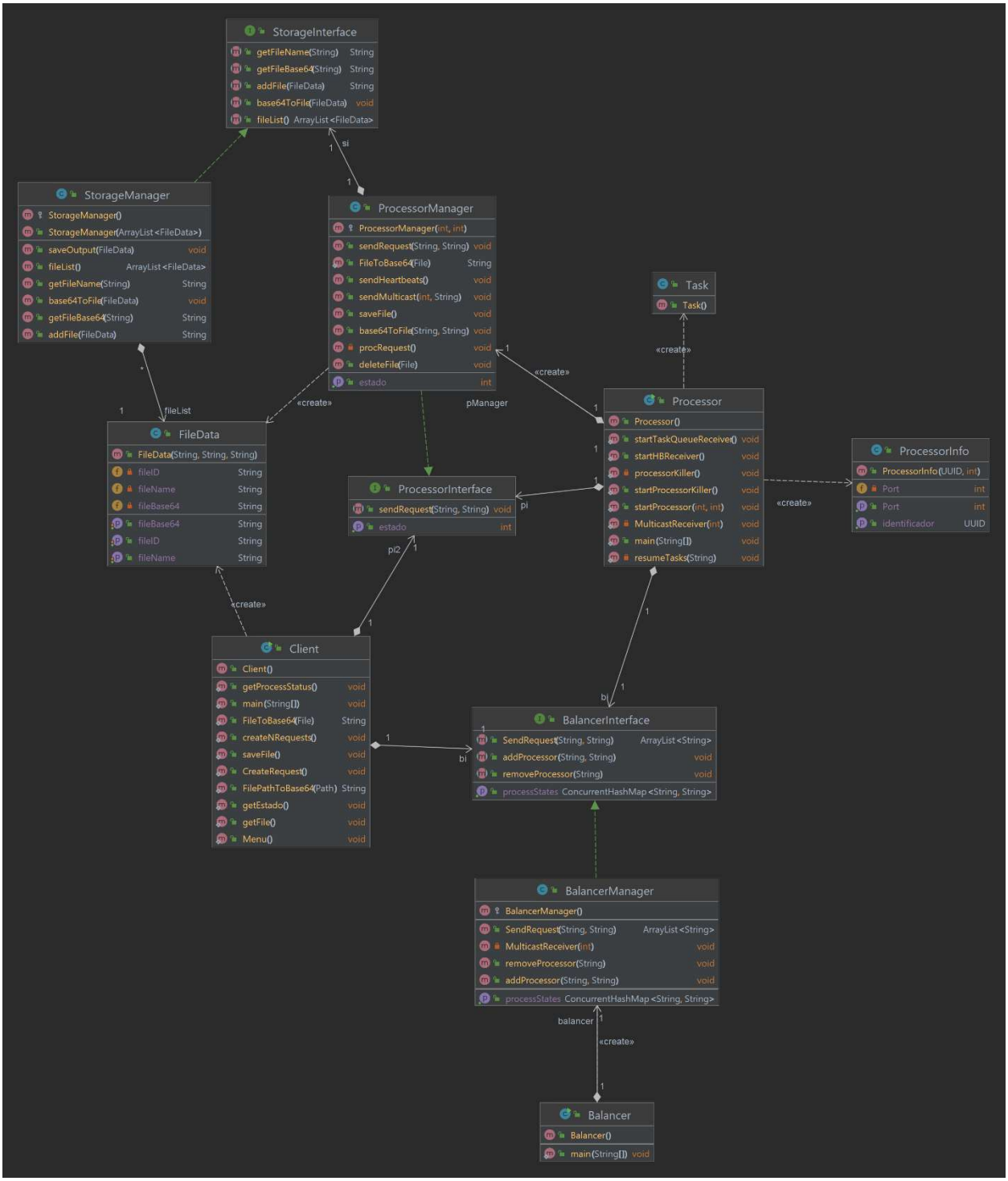
Alexandre Valejo Moreira – 20223

Índice

Introdução

Num mundo cada vez mais rápido e impulsionado pela tecnologia, a gestão eficiente de tarefas é crucial para que se alcance o sucesso na realização das mesmas. Uma forma de alcançar essa eficiência é através do uso de sistemas de distribuição para gerir e executar tarefas de forma organizada e eficiente. Neste relatório, explicaremos o programa em Java em que implementamos um sistema com Processadores, Balanceador, Coordenador e Armazenamento para atingir este objetivo. O programa funciona tendo por base o balanceador, usado para distribuir tarefas entre os processadores, que depois as executam e guardam o output gerado no sistema de armazenamento (*storage*). O coordenador é responsável por receber *heartbeats* dos processadores e geri-los de forma eficiente, de modo a garantir o bom funcionamento do sistema. De uma maneira resumida, ao analisarmos as funcionalidades e a eficácia deste programa, conseguimos descobrir quais os benefícios e possíveis problemas ao utilizar um sistema distribuído.

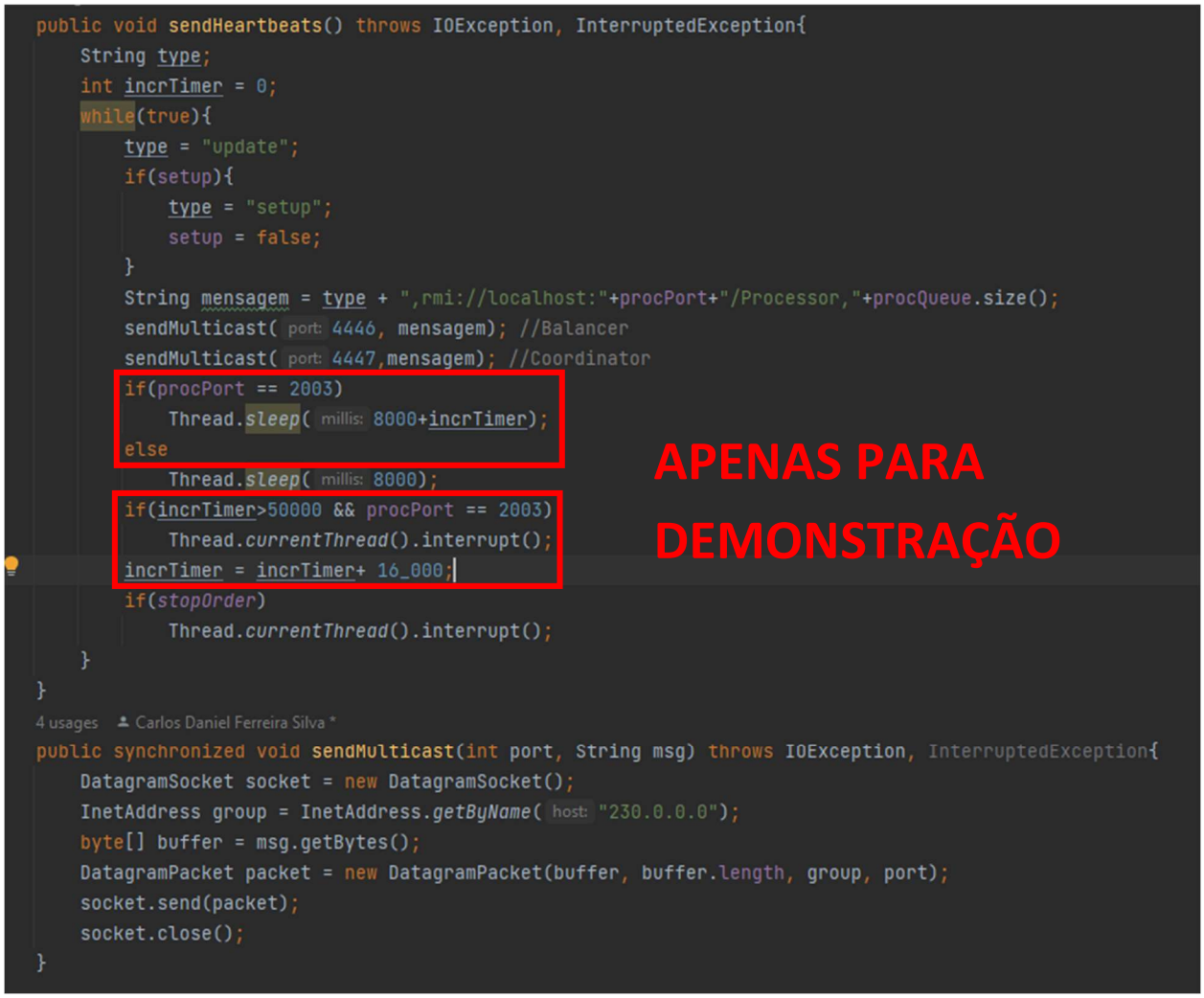
UML


StorageInterface
getFileName(String) String
getFileBase64(String) String
addFile(FileData) String
base64ToFile(FileData) void
fileList() ArrayList<FileData>
si
StorageManager
StorageManager()
StorageManager(ArrayList<FileData>)
saveOutput(FileData) void
fileList() ArrayList<FileData>
getFileName(String) String
base64ToFile(FileData) void
getFileBase64(String) String
addFile(FileData) String
ProcessorManager
ProcessorManager(int, int)
sendRequest(String, String) void
FileToBase64(File) String
sendHeartbeats() void
sendMulticast(int, String) void
saveFile() void
base64ToFile(String, String) void
procRequest() void
deleteFile(File) void
estado int
Task
Task()
«create»
fileList
«create»
pManager
«create»
Processor
Processor()
startTaskQueueReceiver() void
startHBReceiver() void
processorKiller() void
startProcessorKiller() void
startProcessor(int, int) void
MulticastReceiver(int) void
main(String[]) void
resumeTasks(String) void
ProcessorInfo
ProcessorInfo(UUID, int)
Port int
Port int
identificador UUID
«create»
FileData
FileData(String, String, String)
fileID String
fileName String
fileBase64 String
fileBase64 String
fileID String
fileName String
ProcessorInterface
sendRequest(String, String) void
estado int
pI
pI2
«create»
Client
Client()
getProcessStatus() void
main(String[]) void
FileToBase64(File) String
createNRequests() void
saveFile() void
CreateRequest() void
FilePathToBase64(Path) String
getEstado() void
getFile() void
Menu() void
bi
BalancerInterface
SendRequest(String, String) ArrayList<String>
addProcessor(String, String) void
removeProcessor(String) void
processStates ConcurrentHashMap<String, String>
BalancerManager
BalancerManager()
SendRequest(String, String) ArrayList<String>
MulticastReceiver(int) void
removeProcessor(String) void
addProcessor(String, String) void
processStates ConcurrentHashMap<String, String>
balancer
«create»
Balancer
Balancer()
main(String[]) void

Implementação

RNF1

Para a implementação do RNF1, *heartbeats*, foi utilizado o *Multicast.* Na imagem seguinte é mostrada a implementação do *Multicast* no Processor.




```
public void sendHeartbeats() throws IOException, InterruptedException{
    String type;
    int incrTimer = 0;
    while(true){
        type = "update";
        if(setup){
            type = "setup";
            setup = false;
        }
        String mensagem = type + ",rmi://localhost:"+procPort+"/Processor,"+procQueue.size();
        sendMulticast( port: 4446, mensagem); //Balancer
        sendMulticast( port: 4447,mensagem); //Coordinator
        if(procPort == 2003)
            Thread.sleep( millis: 8000+incrTimer);
        else
            Thread.sleep( millis: 8000);
        if(incrTimer>50000 && procPort == 2003)
            Thread.currentThread().interrupt();
        incrTimer = incrTimer+ 16_000;
        if(stopOrder)
            Thread.currentThread().interrupt();
    }
}
4 usages  Carlos Daniel Ferreira Silva *
public synchronized void sendMulticast(int port, String msg) throws IOException, InterruptedException{
    DatagramSocket socket = new DatagramSocket();
    InetAddress group = InetAddress.getByName( host: "230.0.0.0");
    byte[] buffer = msg.getBytes();
    DatagramPacket packet = new DatagramPacket(buffer, buffer.length, group, port);
    socket.send(packet);
    socket.close();
}
```

O processador executa o *sendHeartbeats()* que por sua vez vai enviar, periodicamente (8 em 8 segundos), mensagens por multicast, na *port* 4446 para o Balancer e na *port* 4447 para o Coordenador.

Na parte do Balancer criamos uma *Thread* específica para escutar por Multicasts, como podemos ver na imagem seguinte.

```
protected BalancerManager() throws RemoteException {
    Carlos Daniel Ferreira Silva
    Thread multicastThread = new Thread(new Runnable() {
        Carlos Daniel Ferreira Silva
        public void run() {
            try { MulticastReceiver( port: 4446); } catch (Exception ignored) {}}
    });
    multicastThread.start();
}
```

Podemos ver na imagem seguinte a implementação do MulticastReceiver(), que é responsável por receber as mensagens enviadas por *Multicast* na *port* 4446.

```
private void MulticastReceiver(int port) throws IOException {
    MulticastSocket socket1 = null;
    byte[] buffer = new byte[256];
    socket1 = new MulticastSocket(port);
    InetAddress group = InetAddress.getByName( host: "230.0.0.0");
    socket1.joinGroup(group);

    while (true) {
        DatagramPacket packet1 = new DatagramPacket(buffer, buffer.length);
        socket1.receive(packet1);
        String msg1 = new String(packet1.getData(),  offset: 0, packet1.getLength()); //mensagem recebida
        List<String> qList = Arrays.asList(msg1.split( regex: ","));
        String type = qList.get(0);
        String processor = qList.get(1);
        String queue = qList.get(2);
        if(type.equals("update")){
            if(processorLoad.containsKey(processor)){
                processorLoad.replace(processor, queue);
            }
            System.out.println("Processor:\t"+processor);
            System.out.println("Queue:\t\t"+queue);
            System.out.println("-----------------------------------------------------");
        }
        if(stopOrder)
            Thread.currentThread().interrupt();
    }
}
```

Enquanto a *Thread* está a correr, esta função está num ciclo *while()* a fazer escuta constante. Quando é recebida uma mensagem por *Multicast*, a função faz um *split* que identifica se a mensagem tem o identificador “update”. Caso o tenha vai verificar se o processador na mensagem enviada já está na lista

de processadores do Balancer. Caso esteja, vai atualizar a *ConcurrentHashMap<>* com a informação de carga nova.

RNF4

Para a implementação do RNF4 foram adicionadas algumas funcionalidades na parte do *Multicast.*

Começando pelo Processor, foi feita uma alteração no *sendHeartbeats(),* relativamente ao RNF1.

```
private static boolean setup = true;
```

Criamos uma *flag* global "*setup*" que vai ser *true* por defeito, que vai ser útil para o envio do *heartbeat* de *setup*.

```
public void sendHeartbeats() throws IOException, InterruptedException{
    String type;
    int incrTimer = 0;
    while(true){
        type = "update";
        if(setup){
            type = "setup";
            setup = false;
        }
        String mensagem = type + ",rmi://localhost:"+procPort+"/Processor,"+procQueue.size();
        sendMulticast( port: 4446, mensagem); //Balancer
        sendMulticast( port: 4447,mensagem); //Coordinator
```

Podemos ver pela imagem que existe uma *String type,* que vai ser responsável de armazenar o tipo certo de *heartbeat* (*setup* ou *update*). Quando o *sendHeartbeats()* corre pela primeira vez ele vai obrigatoriamente passar por *type = "setup"* e *setup = false.* Isto permite que o *setup* é sempre enviado por *Multicast* antes de enviar um *update*, que serve para alertar o Coordenador que um processador novo acabou de arrancar.

```java
private static void MulticastReceiver(int port) throws IOException {
    MulticastSocket socket1 = null;
    byte[] buffer = new byte[25600];
    socket1 = new MulticastSocket(port);
    InetAddress group = InetAddress.getByName( host: "230.0.0.0");
    socket1.joinGroup(group);
    if(port == 4447) {
        try {
            bi = (BalancerInterface) Naming.lookup( name: "rmi://localhost:2001/Balancer");
        } catch (NotBoundException | RemoteException | MalformedURLException e) {
            throw new RuntimeException(e);
        }
        while (true) {
            DatagramPacket packet1 = new DatagramPacket(buffer, buffer.length);
            socket1.receive(packet1);
            String msg1 = new String(packet1.getData(),  offset: 0, packet1.getLength());
            List<String> qList = Arrays.asList(msg1.split( regex: ","));
            String type = qList.get(0);
            String processor = qList.get(1);
            String queue = qList.get(2);
            if (type.equals("setup")) {
                processorState.putIfAbsent(processor, 0);
                processorLoad.putIfAbsent(processor, Integer.parseInt(queue));
                bi.addProcessor(processor, queue);
            }
            if (type.equals("update")) {
                processorState.replace(processor, 0);
                processorLoad.replace(processor, Integer.parseInt(queue));
            }
            if(stopOrder)
                Thread.currentThread().interrupt();
        }
```

Na imagem anterior conseguimos ver *o MulticastReceiver()* do Coordenador. Ele vai escutar no *Multicast* por *packets* com informação de tipo *setup* e *update*. Caso seja de setup ele vai adicionar o processador à *ConcurrentHashMap<String, Integer> processorState,* onde o segundo índice vai servir como contador de segundos que o Coordenador não recebe informação do processador. Vai também, no *setup*, adicionar informação sobre a carga do processador (*ConcurrentHashMap<String, Integer> processorLoad*). Por fim, envia um pedido por RMI ao Balancer a pedir a adição do processador à lista de processadores do Balancer com a respetiva carga.

Caso seja do tipo *update* só faz a atualização dos dados e o *refresh* do temporizador.

RNF3

Para a implementação do RNF3 foram feitas várias adições ao Coordenador e *Processor*.

No *Processor* foi implementado o envio de *Multicast* para o Coordenador sempre que recebe um pedido de execução de uma tarefa e a conclusão da mesma.

```
public void sendRequest(String script, String IDFile) throws IOException, InterruptedException {
    if(script==null || IDFile == null)
        return;
    else{
        procQueue.add(taskId+","+script+","+IDFile);
        sendMulticast( port: 4448,  msg: "0,"+taskId+",rmi://localhost:"+procPort+"/Processor,"+script+","+IDFile);
        taskId++;
    }
}
```

```
private void procRequest() throws IOException, InterruptedException{
    System.out.println("["+procId+"]Listening for Process'");
    while(true){
        if(procQueue.iterator().hasNext()) {
            //System.out.println("-------------------["+procId+"]Starting process-------------------");
            String qItem = procQueue.remove();
            List<String> qList = Arrays.asList(qItem.split( regex: ","));
            String id = qList.get(0);
            String script = qList.get(1);
            String IDFile = qList.get(2);
            Files.put(IDFile, script);
            sendMulticast( port: 4448,  msg: "1,"+id+",rmi://localhost:"+procPort+"/Processor");
            //System.out.println(script);
```

Sempre que é recebido um pedido é enviado um *Multicast* com identificador “0”. Quando a tarefa é executada é enviado um *Multicast* com identificador “1”. Esta informação é guardada pelo Coordenador, que vai na essência guardar as tarefas de todos os processadores que estão por executar.

```java
while (true) {
    DatagramPacket packet1 = new DatagramPacket(buffer, buffer.length);
    socket1.receive(packet1);
    String msg1 = new String(packet1.getData(), offset: 0, packet1.getLength());
    List<String> qList = Arrays.asList(msg1.split( regex: ","));
    String status = qList.get(0);
    String taskId = qList.get(1);
    String processor = qList.get(2);
    if (status.equals("0")) {
        String script = qList.get(3);
        String file = qList.get(4);
        Task task = new Task();
        task.id = taskId;
        task.script = script;
        task.file = file;
        if (processorData.containsKey(processor)) {
            processorData.get(processor).add(task);
        } else {
            List<Task> tasks = new ArrayList<>();
            tasks.add(task);
            processorData.put(processor, tasks);
        }
    } else {
        List<Task> tasks = processorData.get(processor);
        tasks.removeIf(t -> t.id.equals(taskId));
    }
    if(stopOrder)
        Thread.currentThread().interrupt();
}
```

Caso o identificador seja “0” significa que é um pedido novo, o qual vai ser gravado no *ConcurrentHashMap<String, List<Task>> processorData*. Caso seja “1” significa que é uma finalização de uma tarefa por parte do Processor, a qual vai ser eliminada da *Map*.

No Coordenador, começamos então por criar uma *Thread* que vai ser responsável por "matar" os *Processors* que não enviaram *heartbeats* nos últimos 30 segundos.

```
public static void startProcessorKiller(){
    new *
    Thread killerThread = new Thread(new Runnable() {
        new *
        public void run() {
            try { processorKiller(); } catch (Exception ignored) {}}
    });
    killerThread.start();
}
```

Na imagem seguinte podemos ver a função *processorKiller()* do Coordenador.

```
private static void processorKiller() throws InterruptedException{
    try {
        bi = (BalancerInterface) Naming.lookup( name: "rmi://localhost:2001/Balancer");
    } catch (NotBoundException | RemoteException | MalformedURLException e) {
        throw new RuntimeException(e);
    }
    while(true){
        Thread.sleep( millis: 1000);
        processorState.forEach((k, v) ->{
            v = v + 1;
            processorState.replace(k, v);
            if(v>30){
                processorState.remove(k);
                try {
                    bi.removeProcessor(k);
                    processorLoad.remove(k);
                    new *
                    Thread resumeTasksThread = new Thread(new Runnable() {
                        new *
                        public void run() {
                            try { resumeTasks(k); } catch (Exception ignored) {}}
                    });
                    resumeTasksThread.start();
                } catch (IOException e) {
                    throw new RuntimeException(e);
                }
            }
            System.out.println("Processor:\t"+k);
            System.out.println("Last HB:\t"+v);
            System.out.println("++++++++++++++++++++++++++++++++++++++++++++++++++");
        });
        if(stopOrder)
            Thread.currentThread().interrupt();
    }
}
```

Na função *while()*, é percorrido o *processorState* e sucessivamente adicionado +1 ao índice do valor. Caso o mesmo passa a ser maior que 30 significa que já não recebe atualizações à 30 segundo, dado pelo facto que o *while()* é executado uma vez por segundo, com auxílio da função *Thread.sleep(1000)*. Se for esse o caso, vai ser removido o Processor do *processorState,do processorLoad* e de seguida envia um pedido de eliminação do Processor da parte do Balancer. Para resumir as tarefas pendentes do processador que já não está a correr, é criada uma *Thread* que vai executar uma função que é responsável pelo envio das mesmas.

```java
private static void resumeTasks(String processor) throws IOException, NotBoundException, InterruptedException {
    String keyWithLowestValue = null;
    int minValue = Integer.MAX_VALUE;

    for (Map.Entry<String, Integer> entry : processorLoad.entrySet()) {
        int value = entry.getValue();
        if (value < minValue) {
            minValue = value;
            keyWithLowestValue = entry.getKey();
        }
    }
    pi = (ProcessorInterface) Naming.lookup(keyWithLowestValue);
    List<Task> tasks = processorData.get(processor);
    System.out.println("$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$");
    System.out.println("Sending "+tasks.size()+" tasks to "+keyWithLowestValue+" from "+processor);
    System.out.println("$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$");
    for (Task task : tasks) {
        String file = task.file;
        String script = task.script;
        System.out.println("Sending Task "+task.id+" to "+keyWithLowestValue);
        pi.sendRequest(script, file);
    }
    Thread.currentThread().interrupt();
}
```

A função começa por percorrer a *ConcurrentHashMap<String, Integer> processorLoad* toda à procura do Processor que tem menos carga. Quando o mesmo é encontrado, é feita uma ligação RMI a esse Processor. De seguida é guardado as tarefas todas pendentes numa Lista, a qual vai ser percorrida e é feito o envio das tarefas para o *Processor* com menos carga.

Conclusão

Ao longo do trabalho surgiram várias dificuldades, atingindo um resultado não perfeito, mas bastante positivo. Conseguindo então finalizar a realização do trabalho prático. As principais dificuldades foram a implementação do RNF9 e RNF11.

Este trabalho ajudou também a desenvolver as nossas competências de investigação, de organização enquanto grupo e na discussão entre cada elemento sobre diferentes pontos de vista.