IPV – Instituto Politécnico de Viseu

ESTGV – Escola Superior de Tecnologia e Gestão de Viseu

Departamento de Informática

Relatório do Projeto de Sistemas Distribuídos

Licenciatura em Engenharia Informática

Realizado em

Sistemas Distribuídos

por

Luís Pais – 20253

Artur Santos – 20251

Daniel Vale – 20246

ESTGV: Carlos Cunha

Viseu, 2022

IPV – Instituto Politécnico de Viseu
ESTGV – Escola Superior de Tecnologia e Gestão de Viseu
Departamento de Informática

Relatório do Projeto de Sistemas Distribuídos

Licenciatura em Engenharia Informática

Realizado em
Sistemas Distribuídos
por

Luís Pais – 20253
Artur Santos – 20251
Daniel Vale - 20246

ESTGV: Carlos Cunha

Viseu, 2022

Índice

Índice de Figuras

1 Introdução

No âmbito da unidade curricular de Sistemas Distribuídos, foi pedido que fosse implementada uma aplicação em Java que tem como principal objetivo, fazer a distribuição da carga imposta pelos diversos pedidos de clientes.

Para que esta distribuição de carga fosse possível, foi implementado um balanceador que distribui os pedidos pelos processadores existentes. Estes pedidos transportam scripts que irão ser executados nos processadores sobre ficheiros previamente armazenados pelo sistema.

Para que estes ficheiros sejam armazenados no sistema, foi implementada uma camada de storage.

Por fim, foi implementado um coordenador, que irá recolher dados sobre os processadores e enviá-los ao balanceador, para que este consiga fazer uma melhor distribuição de carga.

A comunicação entre as diferentes camadas é essencialmente feita por Remote Method Invocation (RMI) e por Multicast.

A metodologia adotada foi SCRUM, ou seja, o projeto foi realizado ao longo de vários sprints, que culminaram neste trabalho final.

2 Arquitetura usada em UML

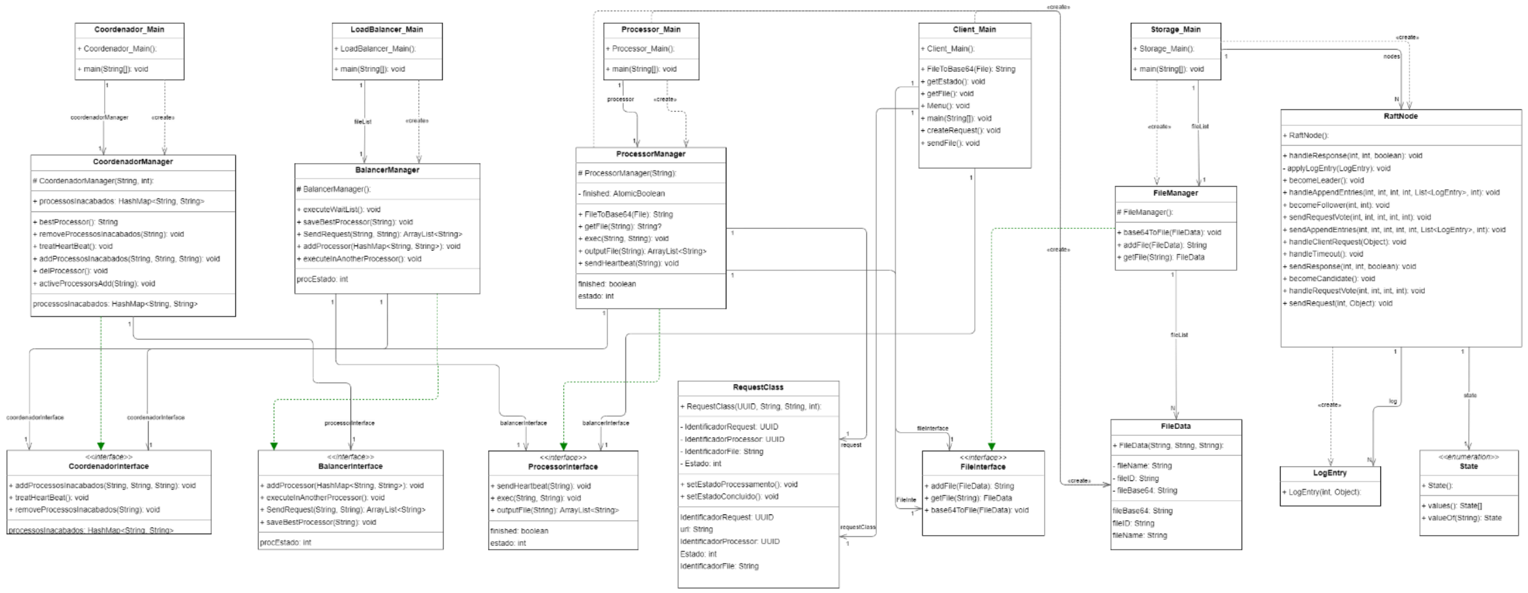


Figura 1 - Arquitetura usada em UML

Na figura 1 está representada a arquitetura da solução em UML. Aqui é possível observar todas as classes criadas e as relações entre elas. Estas foram necessárias para que o projeto fosse concluído com sucesso.

Nota: Embora a imagem possa não ser muito legível, a imagem original está presente na pasta do projeto, onde é possível observá-la com maior facilidade.

3 Implementação

3.1 Storage

Como já foi referido anteriormente, a camada de Storage serve essencialmente para guardar os ficheiros pedidos, sejam estes pedidos realizados pelos clientes ou pelos processadores. Nesta camada existem 4 classes: FileData, FileInterface, FileManager, RaftNode e Storage_Main.

```
public class FileManager extends UnicastRemoteObject implements FileInterface {

    2 usages
    private static final ArrayList<FileData> fileList = new ArrayList<>();

    1 usage  Artur Santos
    protected FileManager() throws RemoteException {
    }

    1 usage  Artur Santos
    public void base64ToFile(FileData f) throws IOException {
        byte[] decodedImg = Base64.getDecoder().decode(f.getFileBase64().getBytes(StandardCharsets.UTF_8));
        Path destinationFile = Paths.get( first: "Storage\\src\\savedFiles", f.getFileName());
        Files.write(destinationFile, decodedImg);
    }

    no usages  Artur Santos +1
    public String addFile(FileData f) throws RemoteException {
        UUID id = UUID.fromString(UUID.nameUUIDFromBytes(String.valueOf(f.getFileBase64()).getBytes()).toString());
        f.setFileID(id.toString());
        fileList.add(f);
        System.out.println(f.getFileID() + " name: " + f.getFileName());  //For testing
        try {
            base64ToFile(f);
        } catch (Exception e) {
            System.out.println("base64ToFile Error: " + e.getMessage() + "\n");
        }
        return id.toString();
    }

    no usages  Artur Santos
    public FileData getFile(String UIDD) {
        for (FileData fileData : fileList) {
            if (UIDD.equals(fileData.getFileID())) {
                return fileData;
            }
        }
        return null;
    }
}
```

Figura 2 - FileManager.java - Storage

Na Figura 2, está presente uma das classes mais importantes da camada Storage. Neste ficheiro, foram criados todos os métodos e objetos necessários para o armazenamento e tratamento de ficheiros.

O método base64ToFile() recebe como parâmetro um objeto da classe FileData, que é composto por “fileID”, ”fileName” e “fileBase64”. Após a receção, descodifica a string “fileBase64”, que

se encontra codificada em base64. Após a descodificação, o ficheiro é guardado na diretoria savedFiles com o nome do ficheiro introduzido pelo utilizador.

O método addFile() atribui um Universal Unique Identifier (UUID) ao ficheiro recebido por parâmetro e adiciona-o à lista de ficheiros (fileList) criada para armazenar os dados de cada ficheiro. De seguida invoca o método base64ToFile() para guardar fisicamente o ficheiro na diretoria savedFiles.

O método getFile() recebe como parâmetro um UUID, e percorre a lista de ficheiros (fileList). Caso encontre o ficheiro com o UUID inserido, devolve um objeto do tipo FileData, para que possa ser tratado posteriormente

3.2 Processor

Na camada Processor, são tratados os pedidos que envolvem a execução de scripts. Nesta camada existem 8 classes: BalancerInterface, CoordenadorInterface, FileData, FileInterface, Processor_Main, ProcessorInterface, ProcessorManager e RequestClass.

```
public synchronized void sendCoordenadorFailConsensus() throws IOException {
    int port = Integer.parseInt(link.substring(16,20));
    if(activeProcessorSize == 1){
        try {
            processors.put(link,port);
            declareConsensus();
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        }
        else{
            try {
                processors.put(link,port);
                DatagramSocket socket = new DatagramSocket();
                InetAddress group = InetAddress.getByName( host: "232.0.0.0");
                String message = "fail "+link;
                byte[] buf = message.getBytes();
                DatagramPacket packet = new DatagramPacket(buf, buf.length, group,  port: 4448);
                socket.send(packet);
                declareConsensus();
                socket.close();
            } catch (RemoteException e) {
                e.printStackTrace();
            } catch (SocketException e) {
                throw new RuntimeException(e);
            } catch (UnknownHostException e) {
                throw new RuntimeException(e);
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        }

}
```

Figura 3 - Método sendCoordenadorFailConsensus() – ProcessorManager.class - Processor

Quando cada processador deteta uma falha no coordenador, o método sendCoordenadorFailConsensus() é chamado. Começa por verificar o número de processadores ativos e caso haja apenas um, é adicionado esse processador e a sua porta a um HashMap que contém o número de processadores em consenso. Caso haja mais que um processador ativo, começa-se por adicionar o próprio processador ao HashMap “processors” (o mesmo referido na situação presente mais acima). De seguida é criada a mensagem “fail”+link, isto é a mensagem irá conter a string “link” e a string e que contém o link do próprio processador. Essa mensagem é enviada por multicast para que possa chegar aos diferentes processadores. Por fim é chamado o método declareConsensus() que irá ser explicado mais à frente.

```java
public synchronized void receiveCoordenadorConsensus() throws IOException {
    Thread t = (new Thread(() -> {
        try {
            MulticastSocket socket = new MulticastSocket( port: 4448);
            InetAddress group = InetAddress.getByName( host: "232.0.0.0");
            socket.joinGroup(group);
            byte[] buf = new byte[256];
            DatagramPacket packet = new DatagramPacket(buf, buf.length);
            socket.receive(packet);
            String received = new String(
                    packet.getData(),  offset: 0, packet.getLength());
            if(received.contains("fail")){
                String linkProcessor = (received.substring( beginIndex: 5));
                Integer port = Integer.valueOf((received.substring(21,25)));
                if(!processors.containsKey(linkProcessor)){
                    processors.put(linkProcessor,port);
                    declareConsensus();
                }
            }
            socket.leaveGroup(group);
            socket.close();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }));
    t.start();
}
```

Figura 4 - Método receiveCoordenadorConsensus() - ProcessorManager.class - Processor

O método receiveCoordenadorConsensus() recebe as mensagens enviadas por multicast pelo método sendCoordenadorFailConsensus(). A mensagem, após ser recebida é lida e verifica-se se a mesma contém a string “fail”, para que possa ser garantido que a mensagem veio do sítio certo. É guardado o link do processador contido na mensagem recebida e caso este ainda não esteja presente no HashMap “processors” é adicionado, juntamente com a sua porta e é chamado o método declareConsensus().

```java
public synchronized void declareConsensus() throws IOException {
    System.out.println("Processor "+processors+" declared consensus");
    if (activeProcessorSize <= 2) {
        if (processors.size() == 1 && activeProcessorSize == 1) {
            System.out.println("Coordenador Failed By Consensus");
        }
        else if (processors.size() == 2 && activeProcessorSize == 2) {
            System.out.println("Coordenador Failed By Consensus");
        }
    } else {
        if (processors.size() > (activeProcessorSize / 2)) {
            System.out.println("Coordenador Failed By Consensus");
        }
    }
}
```

Figura 5 - Método declareConsensus() - ProcessorManager.class - Processor

O método declareConsensus() é chamado pelos métodos presentes anteriormente. Começa-se por mostrar quais os processadores que já declararam consenso (presentes no HashMap "processors"). De seguida é verificado se o número de processadores ativos é menor ou igual a dois, visto que se o número de processadores ativos for igual a 1 e o número de processadores que declararam consenso também for igual a 1 é mostrada a mensagem de consenso – "Coordenador Failed By Consensus". Caso o número de processadores ativos seja 2, só irá ser declarado consenso quando 2 processadores declararem consenso, visto que a maioria tem que ser o número total de processadores. Se o número de processadores ativos for maior do que 2, verifica-se se o número de processadores em consenso é maior do que o número de processadores ativos a dividir por 2 e, caso se verifique, é declarado consenso.

```java
public synchronized void checkAliveCoordenador() throws IOException, NotBoundException {
    Thread t = (new Thread(() -> {
            try {
                MulticastSocket socket = new MulticastSocket( port: 4447);
                InetAddress group = InetAddress.getByName( host: "231.0.0.0");
                socket.joinGroup(group);
                String received = null;
                while (true) {
                    byte[] buf = new byte[256];
                    DatagramPacket packet = new DatagramPacket(buf, buf.length);
                    socket.receive(packet);
                    String encrypted = new String(
                            packet.getData(),  offset: 0, packet.getLength());
                    if (encrypted != null) {
                        char[] chars = encrypted.toCharArray();
                        for(int i = 0; i<chars.length;i++){
                            chars[i] = (char) (chars[i]-10);
                        }
                        received = new String(chars);
                        //System.out.println(encrypted);
                        //System.out.println(received);
                        activeProcessorSize = Integer.parseInt(received.substring( beginIndex: 5));
                        threadStatus = true;
                        lastHeartBeat = Instant.now();
                        processors.clear();
                        break;
                    }
                }
                socket.leaveGroup(group);
                socket.close();
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
    }));
    if(threadStatus){
        t.start();
    }
    else{
        t.interrupt();
    }
}
```

Figura 6 - Método checkAliveCoordenador() - ProcessorManager.class - Processor

Este método recebe mensagens por multicast (vindas do Coordenador) e após a receção começa por desencriptar a mensagem (as mensagens são encriptadas com uma encriptação simples para garantir integridade). Após a desencriptação é colocado o número de processadores ativos numa variável (este número está contido na mensagem), é atualizada a variável "threadStatus" para "true" (este método está numa thread e esta é interrompida caso esta variável esteja a "false"), é guardado o instante deste heartbeat e limpo o HashMap que contém os processadores em consenso de falha.

```java
public synchronized void handleCoordenadorFailure(){
    Thread t = (new Thread(() -> {
        if(threadStatus == true){
            Instant current, interval;
            current = Instant.now();
            interval = Instant.ofEpochSecond(ChronoUnit.SECONDS.between(lastHeartBeat , current));
            if(interval.getEpochSecond() > 30){
                threadStatus = false;
                if(threadStatus == false){
                    System.out.println("Coordenador is down");
                    try {
                        sendCoordenadorFailConsensus();
                    } catch (IOException e) {
                        throw new RuntimeException(e);
                    }
                }
            }
            try {
                Thread.sleep( millis: 1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }));
    if(threadStatus){
        t.start();
    }
    else{
        t.interrupt();
    }
}
```

Figura 7 - Método handleCoordenadorFailure() - ProcessorManager.class - Processor

O método handleCoordenadorFailure(), embora seja interrompido caso a variável "threadStatus" esteja a "false", começa por verificar o estado da mesma e caso este seja "true", guarda numa variável o instante atual. De seguida compara esse valor com o valor guardado na variável "lastHeartBeat" e caso este seja maior que 30 (segundos) coloca a variável "threadStatus" a "false", mostra a mensagem "Coordenador is down" e chama o método sendCoordenadorFailConsensus().

```java
public void exec(String fileID, String script) throws IOException {
    coordenadorInterface.addProcessosInacabados(link,fileID,script);
    threadCount++;
    String filename = getFile(fileID);
    finished.set(false);
    Thread t = (new Thread(() -> {
        try {
            String command = script + " " + filename;

            Process runtimeProcess  = Runtime.getRuntime().exec(command);
            System.out.println("Executing scri: " + command);
            runtimeProcess.waitFor();
            System.out.println("Script executed successfully.");
            File path = new File( pathname: "Storage\\src\\savedFiles\\outfile_" + filename);
            String base64 = FileToBase64(path);
            FileData f = new FileData( fileID: null,  fileName: "outfile_" + filename, base64);
            String ID = FileInte.addFile(f);
            System.out.println("File ID: " + ID);

            coordenadorInterface.removeProcessosInacabados(link);
            finished.set(true);
            threadCount--;
        } catch (IOException | InterruptedException e) {
            throw new RuntimeException(e);
        }
    }));
    t.start();
}
```

Figura 8 - Método exec() - ProcessorManager.class - Processor

O método exec() serve para executar os pedidos enviados pelo cliente. Este recebe como parâmetro um “fileID” e um “script”. Inicialmente começamos por invocar um método do coordenador que serve para adicionar a uma lista os dados para processamento, ou seja, é passado o “link” que o processador usa quando está ligado (por exemplo: “rmi://localhost:2022/processor”), o “fileID” e o “script” recebidos por parâmetro. Isto irá servir para, no caso de não ser possível por parte do processador realizar a execução até ao fim, guardar os dados para poder ser encaminhado para outro processador realizar o processamento. Depois a variável global “threadCount” é incrementada para poder passar ao coordenador o número de threads a correr. De seguida é executada a função getFile() que, dado o ID, retorna o nome do ficheiro. De seguida é colocada a variável global “finished” a “false” para que o balanceador possa saber o estado da execução do processador (se estiver a “false” não está terminada, se estiver a “true” está).

Após estes processos é iniciada uma thread, e é executado o comando que ordena a execução do script sobre o ficheiro inseridos. O processador espera que o comando seja executado e após essa execução adiciona o novo ficheiro à camada Storage (processo idêntico ao realizado pelo cliente). Por fim o processo é retirado da lista, a variável “finished” é colocada a “true” e a variável “threadCount” é decrementada. Todos os processos pós execução servem para

indicar às diferentes instâncias com quem o processador comunica que o processo foi terminado.

```java
public ArrayList<String> outputFile(String fileID) throws IOException {
    try {
        String filename = getFile(fileID);
        ArrayList<String> outputLines = new ArrayList<>();
        String file = "Storage\\src\\savedFiles\\outfile_" + filename;
        BufferedReader br = new BufferedReader(new FileReader(file));
        String line;

        while ((line = br.readLine()) != null) {
            outputLines.add(line);
        }
        return outputLines;
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

Figura 9 - Método outputFile() - ProcessorManager.class - Processor

Na figura 9 está presente o método outputFile(). Este método tem como objetivo principal devolver ao cliente o ficheiro output criado após a execução do script. Recebe como parâmetro um "fileID" que será usado para obtenção do nome do ficheiro (filename). De seguida é criada uma ArrayList que irá guardar, linha a linha, o conteúdo do ficheiro de ouput. Após este processo estar concluído, o output é retornado.

```java
public void sendHeartbeat(String multicastMessage) throws RemoteException {
    Thread t = (new Thread(() -> {
        try {
            char[] chars = multicastMessage.toCharArray();
            for(int i = 0; i< chars.length; i++){
                chars[i] = (char) (chars[i]+10);
            }
            String encryptedMessage = new String(chars);

            DatagramSocket socket = new DatagramSocket();
            InetAddress group = InetAddress.getByName( host: "230.0.0.0");
            byte[] buffer = encryptedMessage.getBytes();
            DatagramPacket packet = new DatagramPacket(buffer, buffer.length, group,  port: 4446);
            socket.send(packet);
            socket.close();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }));
    t.start();
}
```

Figura 10 - Método sendHeartbeat() - ProcessorManager.class - Processor

Na figura 10 está presente o método sendHeartBeat(). Este serve para enviar por multicast informações sobre o processador em questão. Começa por receber por parâmetro uma String, encripta-a e transforma-a em bytes para a enviar por multicast.

```java
final public Runnable processorInfo = () -> {
    final HashMap<String, Integer> processsorInfo = new HashMap<>();
    processsorInfo.put(link, threadCount);

    try {
        sendHeartbeat(processsorInfo.toString());
        checkAliveCoordenador();
        handleCoordenadorFailure();
    } catch (RemoteException e) {
        throw new RuntimeException(e);
    } catch (NotBoundException e) {
        throw new RuntimeException(e);
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
};

1 usage
final public Runnable runnableReceive = () -> {
    try {
        receiveCoordenadorConsensus();
    } catch (RemoteException e) {
        throw new RuntimeException(e);
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
};
```

Figura 11 - Métodos proceessorInfo() e runnableReceive() - ProcessorManager.class - Processor

Os métodos presentes na figura 11 são ambos do tipo Runnable. O método processorInfo() guarda num HashMap o link do processador em questão e a variável “threadCount”. De seguida são invocados os métodos sendHeartbeat(), checkAliveCoordenador() e handleCoordenadorFailure(). No caso do método sendHeartBeat(), é-lhe passado o HashMap referido anteriormente. Este método (processorInfo()) é executado de 5 em 5 segundos na função main(). O método runnableReceive() invoca apenas o método receiveCoordenadorConsensus() e é executado de 100 em 100 milissegundos na função main().

3.3 Coordenador

A camada coordenador é responsável por receber os dados de todos os processadores, tratá-los e enviá-los para o Balanceador. Nesta camada existem 4 classes: BalancerInterface, Coordenador_Main, CoordenadorInterface e CoordenadorManager.

```
public void treatHeartBeat() throws RemoteException {
    byte[] buf = new byte[256];
    Thread t = (new Thread(() -> {
        try {
            MulticastSocket socket = new MulticastSocket( port: 4446);
            InetAddress group = InetAddress.getByName( host: "230.0.0.0");
            socket.joinGroup(group);
            while (true) {
                DatagramPacket packet = new DatagramPacket(buf, buf.length);
                socket.receive(packet);
                String encrypted = new String(
                        packet.getData(),  offset: 0, packet.getLength());
                char[] chars = encrypted.toCharArray();
                for(int i = 0; i<chars.length;i++){
                    chars[i] = (char) (chars[i]-10);
                }
                String received = new String(chars);
                activeProcessorsAdd(received);
                if ("end".equals(received)) {
                    break;
                }
            }
            socket.leaveGroup(group);
            socket.close();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }));
    t.start();
}
```

Figura 12 - método treatHeartBeat() - CoordenadorManager - Coordenador

Na figura 12 está presente o método tratHeartBeat(). Este é responsável por receber por multicast as informações enviadas por cada processador. Após a receção do pacote de dados converte-o em String. Após isso desencripta a informação e invoca o método activeProcessorsAdd(), para que este trate os dados recebidos.

```java
public void sendAliveMessage(){
    Thread t = (run() -> {
            while (true){
                String message = "alive"+activeProcessors.size();
                try {
                    char[] chars = message.toCharArray();
                    for(int i = 0; i< chars.length; i++){
                        chars[i] = (char) (chars[i]+10);
                    }
                    String encryptedMessage = new String(chars);
                    DatagramSocket socket = new DatagramSocket();
                    InetAddress group = InetAddress.getByName( host: "231.0.0.0");
                    byte[] buf = encryptedMessage.getBytes();
                    DatagramPacket packet = new DatagramPacket(buf, buf.length, group,  port: 4447);
                    socket.send(packet);
                    socket.close();
                } catch (IOException  e) {
                    throw new RuntimeException(e);
                }
            }
    });
    t.start();
}
```

Figura 13 – Método sendaliveMessage() – ProcessorManager.class - Processor

O método sendAliveMessage() começa por criar uma mensagem que contém a string “alive” e o número de elementos presentes na lista “activeProcessors” (número de processadores ativos). Após a criação da mensagem, esta é encriptada e enviada por multicast. Com esta função, os processadores existentes conseguem saber se o coordenador se encontra ativo e o número de processadores existentes.

```java
public void activeProcessorsAdd(String received) throws RemoteException {
    String receivedToSplit = received.substring( beginIndex: 1);
    String[] arrofreceived = receivedToSplit.split( regex: "=", limit: 2);
    String port = arrofreceived[0];
    String threadsToSplit = arrofreceived[1];
    String[] arrofthreads = threadsToSplit.split( regex: "}", limit: 2);
    String threads = arrofthreads[0];

    synchronized (activeProcessorsToSend) {
        activeProcessorsToSend.put(port, threads);
    }

    String date = String.valueOf(LocalTime.now());
    synchronized (activeProcessors) {
        activeProcessors.put(port, date);
    }
    balancerInterface.addProcessor(activeProcessorsToSend);

    String bestProcessor = bestProcessor();

    balancerInterface.saveBestProcessor(bestProcessor);

    // System.out.println("Processadores ativos: ");
    //System.out.println(activeProcessors);
}
```

Figura 14 - método activeProcessorsAdd() - CoordenadorManager - Coordenador

Na figura 14 é mostrado o método activeProcessorsAdd(). Este tem como principal função o tratamento dos dados recebidos por multicast. Quando o método é invocado, é-lhe passado por parâmetro os dados recebidos. De seguida procede-se ao tratamento da String para que se possa usar os dados recebidos. Inicialmente guardamos na lista "activeProcessorsToSend" o link do processador e o número de threads que estão a ocorrer no mesmo. Esta lista será enviada para o Balanceador. De seguida guardamos noutra lista (activeProcessors) o link do processador e a data e hora em que a função é executada (visto que é invocada pela thread, é executada sempre que são recebidos dados por multicast). Isto servirá para mais tarde ser possível identificar os processadores que pararam de executar. De seguida é invocada uma função do Balanceador e é-lhe passado como parâmetro a lista "activeProcessorsToSend". Após este passo, é guardado numa variável o resultado da execução de uma função que retorna o processador com menor número de threads a correr. Por fim é chamada uma função do Balanceador que guarda a variável obtida no passo anterior.

```
public void delProcessor() throws RemoteException, RuntimeException, InterruptedException {
    Bidiana
    Thread t = (new Thread() {
        Bidiana
        public void run() {
            while (true) {
                String date = String.valueOf(LocalTime.now());
                SimpleDateFormat format = new SimpleDateFormat( pattern: "HH:mm:ss");
                Date datenowConverted = null;

                try {
                    datenowConverted = format.parse(date);
                } catch (ParseException e) {
                    throw new RuntimeException(e);
                }

                synchronized (activeProcessors) {
                    if (!activeProcessors.isEmpty()) {
                        List keys = new ArrayList(activeProcessors.keySet());
                        for (int i = 0; i < keys.size(); i++) {
                            Object obj = keys.get(i);
                            String ultimadataRequest = activeProcessors.get(obj);

                            Date dateOfRequConverted = null;
                            try {
                                dateOfRequConverted = format.parse(ultimadataRequest);
                            } catch (ParseException e) {
                                throw new RuntimeException(e);
                            }
                            long difference = datenowConverted.getTime() - dateOfRequConverted.getTime();
                            if (difference > 30000) {
                                activeProcessors.remove(obj);
                                activeProcessorsToSend.remove(obj);
                                String bestProcessor = bestProcessor();

                                try {
                                    balancerInterface.saveBestProcessor(bestProcessor);
                                } catch (RemoteException e) {
                                    throw new RuntimeException(e);
                                }
                                System.out.println("O processador " + obj + " já não está ativo");

                                try {
                                    balancerInterface.addProcessor(activeProcessors);
                                    System.out.println("Verificar se Executar em outro processador");
                                    System.out.println(processosInacabados);
                                    balancerInterface.executeInAnotherProcessor();

                                } catch (RemoteException e) {
                                    throw new RuntimeException(e);
                                } catch (IOException e) {
                                    throw new RuntimeException(e);
                                } catch (InterruptedException e) {
                                    throw new RuntimeException(e);
                                }
                            }
                        }
                    }
                }
                try {
                    Thread.sleep( millis: 5000);
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }
        }
    });
    t.start();
}
```

Figura 15 - Método delProcessor() - ProcessorManager - Coordenador

O método presente na figura 15 (delProcessor()) tem como principal função eliminar os processadores da lista de processadores ativos, desde que, claro, estes não se encontrem ativos.

Como é possível observar foi criada uma thread que corre de 5 em 5 segundos. Inicialmente começa por guardar a data e hora atual numa variável. Depois é criada uma variável com a formatação de data desejada. Após isto a data é formatada e guardada numa nova variável. Após estes passos iniciais, verificamos se a lista "activeProcessors" está vazia. Caso não esteja, percorremos todos os seus elementos e guardamos numa variável a data e hora do último heartbeat. Após isso a data é convertida no formato desejado e é realizado um cálculo (hora atual subtraída pela hora do último heartbeat) e caso esse cálculo dê mais de 30 segundos (que neste caso significa que o último heartbeat foi há mais de 30 segundos), os processadores em causa são considerados inativos e são removidos das listas "activeProcessors" e "activeProcessorsToSend". Para além disso, é atualizado o valor do melhor processador e a lista presente no Balanceador. Por fim, é chamada uma função do balanceador para que este, caso haja processos (execuções) por concluir, ordene a execução noutro processador que esteja ativo.

```
public String bestProcessor() {
    int low = Integer.MAX_VALUE;
    String port = null;
    if (!activeProcessors.isEmpty()) {
        List keys = new ArrayList(activeProcessorsToSend.keySet());
        for (int i = 0; i < keys.size(); i++) {
            Object obj = keys.get(i);
            int value = Integer.parseInt(activeProcessorsToSend.get(obj));
            if (value < low) {
                low = value;
                port = obj.toString();
            }
        }
    }
    return port;
}

no usages  Bidarra +1
public void addProcessosInacabados(String link, String fileID, String script) throws RemoteException {
    processosInacabados.put(link, fileID + " " + script);
    //System.out.println("Processos Inacabados: ");
    //System.out.println(processosInacabados);
}
no usages  Bidarra +1
public void removeProcessosInacabados(String link) throws RemoteException {
    processosInacabados.remove(link);
    //System.out.println("Processos Inacabados: ");
    //System.out.println(processosInacabados);
}
no usages  Bidarra +1
public HashMap<String,String> getProcessosInacabados() throws RemoteException {
    //System.out.println("Processos Inacabados get: ");
    //System.out.println(processosInacabados);
    return processosInacabados;
}
```

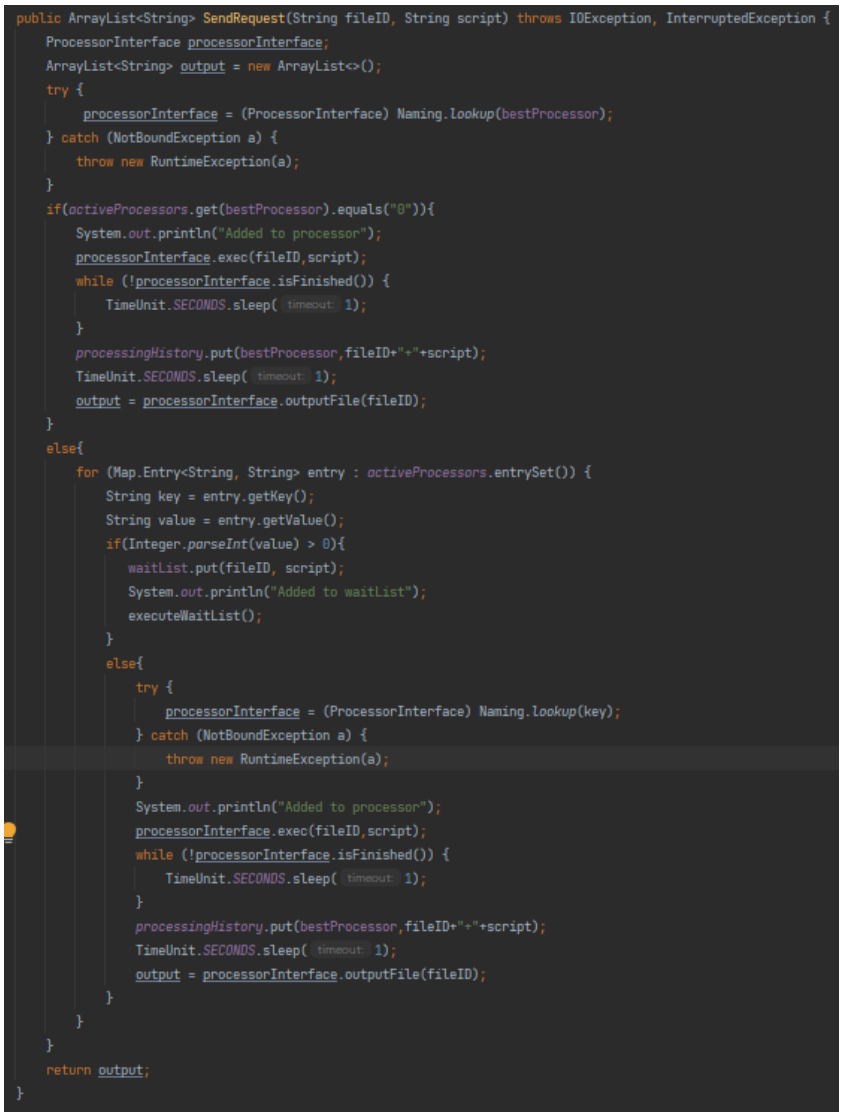
Figura 16 - Métodos bestProcessor(), addProcessosInacabados(),removeProcessosInacabados() e getProcessosInacabados() - CoordenadorManager - Coordenador

O método bestProcessor() presente na figura 16, tem como objetivo, calcular qual o melhor processador, tendo por base o número de threads que está a ocorrer em cada um deles. O algoritmo irá verificar qual é o processador que tem menos threads a correr e, com base nisso retorna o link do processador.

Os métodos addProcessosInacabados(), removeProcessosInacabados() e getProcessosInacabados() foram criados para a interação de outras camadas com a lista "processosInacabados". No caso do método addProcessosInacabados(), são-lhe passados como parâmetros o link que o processador usa, o "fileID" e o "script", e estes irão ser adicionados à lista. No método removeProcessosInacabados() é-lhe passado como parâmetro o link que o processador usa e tendo em conta este link, é removido objeto que tem esse link como chave. Por fim, no método getProcessosInacabados() apenas é retornada a lista "processosInacabados".

3.4 LoadBalancer

A camada LoadBalancer serve para fazer a ligação entre cliente e servidor, isto é, reencaminha os pedidos do cliente para o melhor servidor, de modo que estes possam ser executados mais rapidamente. Esta camada é composta por 7 classes: BalancerInterface, BalancerManager, CoordenadorInterface, FileData, LoadBalancer_Main, ProcessorInterface e RequestClass.



```
public ArrayList<String> SendRequest(String fileID, String script) throws IOException, InterruptedException {
    ProcessorInterface processorInterface;
    ArrayList<String> output = new ArrayList<>();
    try {
         processorInterface = (ProcessorInterface) Naming.lookup(bestProcessor);
    } catch (NotBoundException a) {
        throw new RuntimeException(a);
    }
    if(activeProcessors.get(bestProcessor).equals("0")){
        System.out.println("Added to processor");
        processorInterface.exec(fileID,script);
        while (!processorInterface.isFinished()) {
            TimeUnit.SECONDS.sleep( timeout: 1);
        }
        processingHistory.put(bestProcessor,fileID+"+"+script);
        TimeUnit.SECONDS.sleep( timeout: 1);
        output = processorInterface.outputFile(fileID);
    }
    else{
        for (Map.Entry<String, String> entry : activeProcessors.entrySet()) {
            String key = entry.getKey();
            String value = entry.getValue();
            if(Integer.parseInt(value) > 0){
               waitList.put(fileID, script);
               System.out.println("Added to waitList");
               executeWaitList();
            }
            else{
                try {
                    processorInterface = (ProcessorInterface) Naming.lookup(key);
                } catch (NotBoundException a) {
                    throw new RuntimeException(a);
                }
                System.out.println("Added to processor");
                processorInterface.exec(fileID,script);
                while (!processorInterface.isFinished()) {
                    TimeUnit.SECONDS.sleep( timeout: 1);
                }
                processingHistory.put(bestProcessor,fileID+"+"+script);
                TimeUnit.SECONDS.sleep( timeout: 1);
                output = processorInterface.outputFile(fileID);
            }
        }
    }
    return output;
}
```

Figura 17 - Método SendRequest() - BalancerManager - LoadBalancer

O método presente na figura 17, tem como principal objetivo o envio de pedidos realizados pelo cliente para o processador. Inicialmente é feita a conexão com o melhor processador. Após esta conexão, é verificado se o melhor processador tem alguma thread a correr. Caso não tenha é invocada a função de execução da camada processador e é retornado o output e devolvido ao cliente. Caso o processador tenha processos a decorrer, a lista de processadores ativos é percorrida e caso haja mais processadores na lista, é verificado se cada um deles tem alguma thread a correr. Caso tenha, é adicionado a uma lista de espera (que guarda o “fileID” e o

"script") e é ativada uma thread que irá verificar constantemente se já há processadores disponíveis. Caso não tenha nenhuma execução a correr, liga-se a esse processador e encaminha o pedido para ele. Por fim, é devolvido o output ao cliente.

```
public void executeWaitList() throws IOException, InterruptedException {
    Thread t = (new Thread(() -> {
        if (waitList.size() > 0) {
            ArrayList<String> execution = null;
            while (true){
                for (Map.Entry<String, String> entry : waitList.entrySet()) {
                    String key = entry.getKey();
                    String value = entry.getValue();
                    try {
                        execution = SendRequest(key, value);
                    } catch (IOException e) {
                        throw new RuntimeException(e);
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                    if(execution != null){
                        waitList.remove(key);
                        System.out.println("Removed from waitList");
                    }
                }
            }
        }
    }));
}
```

Figura 18 – Método executeWaitList() - BalancerManager - Balanceador

Na figura 18 está presente o método executeWaitList(). Este é ativado pelo método da figura 9 e quando isso ocorre ele mantém-se a correr numa thread. Neste caso começamos por verificar se há elementos na lista de espera e, caso exista, são percorridos todos os elementos da lista e o método SendRequest() é executado. Caso o output da execução do mesmo seja diferente de "null", ou seja, o pedido já foi encaminhado para o processador e recebido, o objeto é removido da lista de espera.

```java
public void executeInAnotherProcessor() throws IOException, InterruptedException {
    try {
        coordenadorInterface = (CoordenadorInterface) Naming.lookup( name: "rmi://localhost:2050/coordenador");
    } catch (NotBoundException a) {
        throw new RuntimeException(a);
    } catch (MalformedURLException e) {
        throw new RuntimeException(e);
    }
    System.out.println("Executing in another processor");
    HashMap<String, String> lista = coordenadorInterface.getProcessosInacabados();
    System.out.println("Lista de processos inacabados: " + lista);
    if (!lista.isEmpty()) {
        System.out.println("Lista não está vazia");
        ProcessorInterface processorInterface;
        ArrayList<String> output = new ArrayList<>();

        List keys = new ArrayList(lista.keySet());
        for (int i = 0; i < keys.size(); i++) {
            System.out.println("A executar o processo: " + keys.get(i));
            Object obj = keys.get(i);
            lista.get(obj);
            String[] parts = lista.get(obj).split( regex: "\\+");
            String fileID = parts[0];
            System.out.println("FileID: " + fileID);
            String script = parts[1];
            System.out.println("Script: " + script);
            SendRequest(fileID, script);
            lista.remove(obj);
            coordenadorInterface.removeProcessosInacabados(obj.toString());
        }
    }
}
```

Figura 19 - Método executeInAnotherProcessor() - BalancerManager - Balanceador

Na figura 19 está presente o método executeInAnotherProcessor(). Este é invocado pelo Coordenador quando um processador falha. Inicialmente, o método guarda a lista de processos inacabados numa variável. De seguida verifica se esta lista tem algum objeto. Caso tenha, trata os dados de forma a obter o “fileID” e o “script” a executar. Após esse tratamento é invocado o método SendRequest() para encaminhar os pedidos para um novo processador. Por fim, após a execução desse método, o objeto é removido da lista de processos inacabados.

3.5 Client

A camada Client, tem como principal objetivo fazer pedidos para as diferentes camadas do sistema. Esta camada conta com 5 classes: BalancerInterface, Client_Main, FileData, FileInterface e RequestClass.

```
public static void sendFile() {
    try {
        File path;
        String ID;
        System.out.println("Insira o caminho do ficheiro:");
        String toPath = input.next().replace( target: "\"",  replacement: "");
        path = new File(toPath);
        String base64 = FileToBase64(path);
        String fileName = path.getName();
        FileData f = new FileData( fileID: null, fileName, base64);
        ID = fileInterface.addFile(f);
        System.out.println("File ID: " + ID);
    } catch (RemoteException e) {
        System.out.println(e.getMessage());
    } catch (Exception e) {
        e.printStackTrace();
    }
}

1 usage   Bidarra +1
public static void getFile() throws IOException {
    String ID;
    System.out.println("ID:");
    ID = input.next();
    FileData f = fileInterface.getFile(ID);
    if (f == null) {
        System.out.println((Object) null);
    } else {
        System.out.println("Nome do ficheiro: (" + f.getFileName() + ")");
    }
}
```

Figura 20 - Métodos sendFile() e getFile() - Client_Main - Client

Os métodos presentes na figura 20 são invocados quando são selecionados num menu inicial. O método sendFile() serve para enviar um ficheiro para a camada Storage. Neste caso é pedido ao utilizador que introduza o caminho onde se encontra o ficheiro. Depois do utilizador inserir o caminho, o ficheiro é codificado em Base64, e usando o método getName(), é recolhido o nome do ficheiro. Depois disto, é criado um objeto FileData, onde é guardado o nome do ficheiro e a sua codificação em Base64. É de notar que existe um campo preenchido a “null”, isto ocorre porque esse campo irá ser preenchido pela camada storage, quando é invocado o método addFile(). O output (UUID atribuído ao ficheiro) dessa invocação é guardado numa

variável e é mostrado ao utilizador. O método getFile() é invocado quando o cliente seleciona uma opção no menu. Este método serve para, dado um ID inserido pelo o utilizador, retornar o nome do ficheiro, caso este exista.

```
public static void createRequest() throws IOException, InterruptedException {
    String fileID;
    System.out.println("Insira o ID do ficheiro a enviar (O ficheiro precisa de estar em Storage)");
    fileID = input.next();
    System.out.println("Insira o nome do Script que deseja executar");
    String script = input.next();
    String scriptQuotationMarks = "\"" + script + "\"";
    try {
        ArrayList<String> outputContent = balancerInterface.SendRequest(fileID, scriptQuotationMarks);
        for (String s : outputContent) {
            System.out.println(s);
        }
    }catch (Exception e){
        System.out.println("Erro: " + e.getMessage());
    }
}
```

Figura 21 - Método createRequest() - Client_Main - Client

O método createRequest(), que está presente na figura 21, tem como objetivo enviar um pedido para o balanceador para que este o reencaminhe para um processador. Neste caso é pedido ao utilizador que insira o ID do ficheiro e o script que ele pretende. De seguida o pedido é enviado ao balanceador e passando o “fileID” e o “script” por parâmetro. O output é guardado numa lista e caso seja possível, é mostrado ao cliente.

4 Conclusão

Após a realização deste projeto, o grupo conseguiu aprofundar bastante o conhecimento acerca de Sistemas Distribuídos, e a forma como estes se comportam para que haja uma melhor performance, menos falhas e melhor distribuição de carga dentro de diversos sistemas.

O grupo pensa que o projeto foi realizado com sucesso, visto que se conseguiu dar resposta à maioria dos requisitos funcionais e não funcionais impostos. A solução idealizada pelo grupo é satisfatória pelo motivo mencionado anteriormente, embora possa não ser a mais eficaz. Isto é, embora o sistema funcione e realize as funções que lhe compete, algumas das formas que foram encontradas para resolver certos problemas, podem gastar mais recursos do que o pretendido.

Embora o projeto tenha sido concluído com sucesso, o grupo confrontou-se com algumas adversidades e limitações. Estas, assentam essencialmente no facto de o Sistema ter de ser implementado nas máquinas dos alunos, ou seja, para poder testar as funcionalidades implementadas, os recursos usados pelos programas foram bastante altos, o que pode por vezes ter tornado certos processos mais demorados. Para além desta limitação, podemos ainda referir que para testar um sistema deste género, para os testes serem o mais fiéis possíveis, teríamos de estar na presença de processos com escalas muito superiores, isto é, a capacidade de testagem em localhost e com o recurso a uma máquina torna os testes mais simples e menos reais. Por este motivo, para fazer uma testagem mais fidedigna, teríamos de realizar testagem em massa, com múltiplos clientes, processadores, etc.

Para uma possível implementação no futuro, talvez pudessem ser resolvidos alguns dos problemas de eficácia referidos anteriormente. Para isto poderíamos tentar distribuir melhor a carga atribuída ao coordenador, isto é, o coordenador realiza muitas funções no domínio da gestão dos processadores utilizados, e talvez distribuindo a carga por processadores e balanceador e não centralizando tantas operações no coordenador, fosse conseguida uma maior eficácia e performance.