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

Unidade Curricular: Sistemas Distribuídos

Relatório Relativo ao Projeto Final SDt

Tema: Projeto Final SDt

Realizado por: Hugo Marques – 18989

Daniel Moreira – 21096

Mara Lopes – 19506

Rui Ribeiro - 17311

Viseu, 2022

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

Relatório Relativo ao Projeto Final SDt

Curso de Licenciatura em Engenharia Informática

Unidade Curricular de Sistemas Distribuídos

Relatório Relativo ao Projeto Final SDt

Ano Letivo 2022/23

Viseu, 2022

ÍNDICE

Índice de Figuras

1. Introdução

Este projeto final foi desenvolvido no âmbito da disciplina de Sistemas Distribuídos. Durante a realização deste, existiu vários sprints onde se tinha de implementar requisitos funcionais e não funcionais.

O tema deste projeto consistiu em desenvolver um sistema que conseguiu-se escalar a gestão/processamento da execução de diferentes processos. Neste, foi implementado o Cliente, o Servidor, o Balanceador e os Processadores.

O Balanceador é quem vai ser responsável pela distribuição de carga, pelos Processadores. O Cliente envia ficheiros para a camada *storage* e para a camada de processamento, submete os pedidos que "levam" o script e o identificador do ficheiro aos Processadores, para que estes o possam executar.

Este documento está organizado em 7 capítulos que se seguem a esta introdução.

No 2º capítulo apresenta-se a arquitetura da solução em UML.

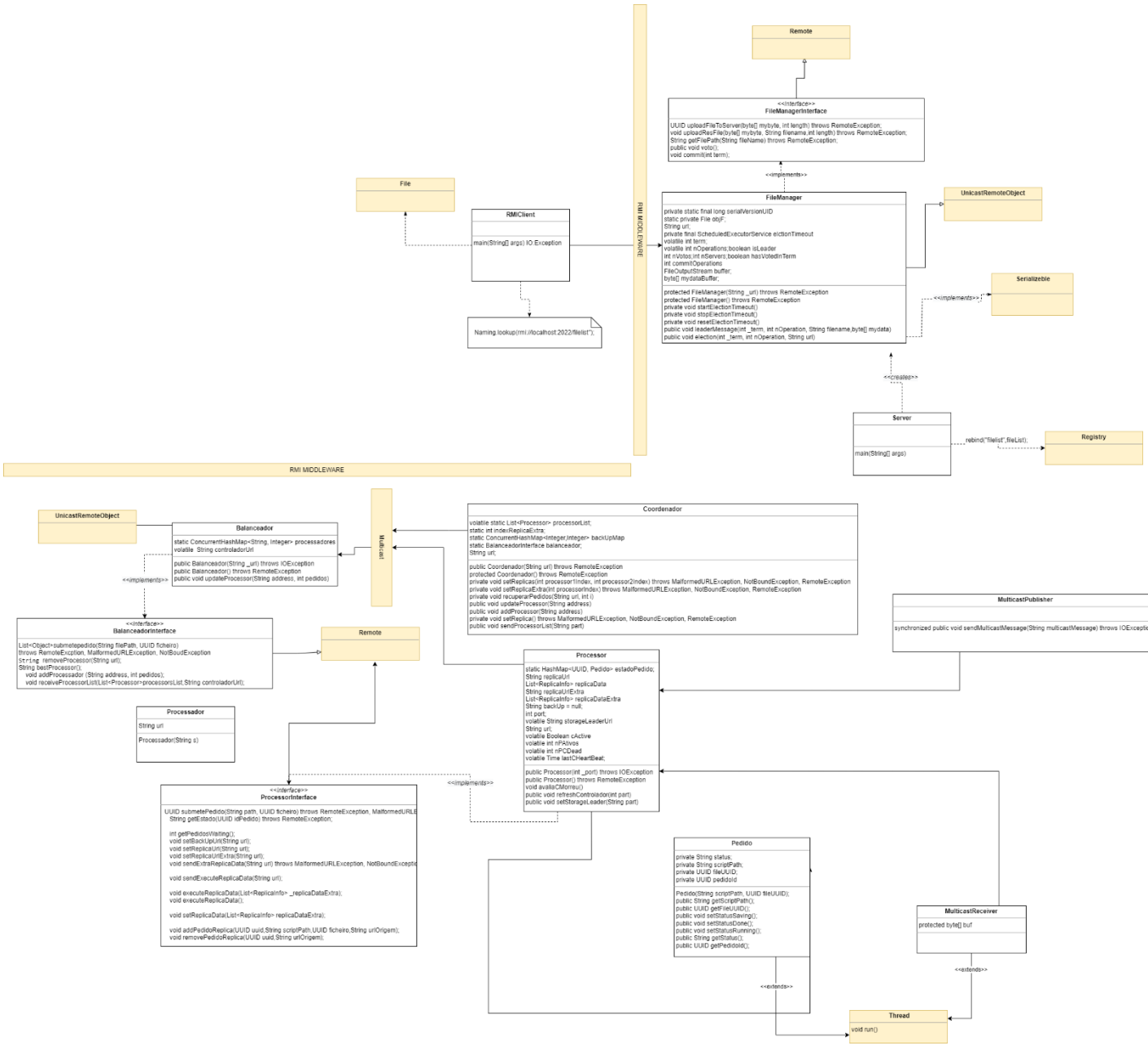
No 3º, 4º e 5º capítulos apresentam-se o desenvolvimento dos *sprints* 1, 2 e 3, onde se apresentam as resoluções dos requisitos funcionais.

No 6º e 7º capítulos apresentam-se o desenvolvimento dos *sprints* 4, 5 e 6, onde se apresentam as resoluções dos requisitos não funcionais.

No 8º capítulo apresentam-se as dificuldades que se tiveram ao longo do desenvolvimento do projeto.

Termina-se com o 9º capítulo, onde se apresentam as conclusões deste trabalho.

2. Arquitetura da solução em UML



3. Sprint 1

3.1. Descrição do Sprint 1

rf1. O cliente submete um pedido armazenamento de dados no processo líder da camada de *storage*, e recebe um identificador para os dados.

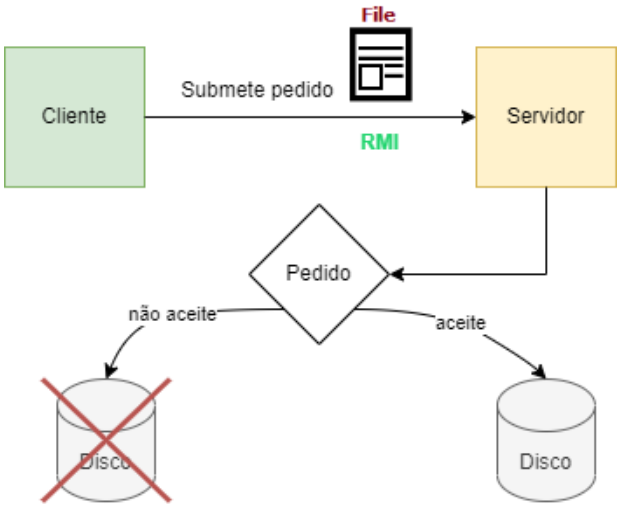


Figura 1 : Diagrama - Requisito Funcional 1

3.1.1. Resolução do Requisito Funcional 1

Inicialmente, criou-se dois projetos, o Cliente Servidor de Armazenamento.

O Cliente vai submeter um pedido ao Servidor para guardar o ficheiro, e o Servidor por sua vez recebe o pedido, e senão surgir nenhum erro, armazena o ficheiro.

```
2 usages   HugoNeves0808
private boolean saveFile(byte[] mydata, String filename,int length){
    try {
        File serverpathfile = new File(filename);
        FileOutputStream out=new FileOutputStream(serverpathfile);

        out.write(mydata);
        out.flush();
        out.close();
        return true;
    } catch (IOException e) {
        e.printStackTrace();
    }
    return false;
}
```

Figura 2 : Função - Guardar ficheiro submetido pelo Cliente

4. Sprint 2

4.1. Descrição do Sprint 2

rf2. O cliente submete um pedido de processamento ao balanceador de carga, que será encaminhado para um dos processadores e recebe um identificador para o pedido, juntamente com o identificador do processador responsável pelo processamento.

rf3. O cliente verifica o estado do pedido a qualquer momento junto do processador.

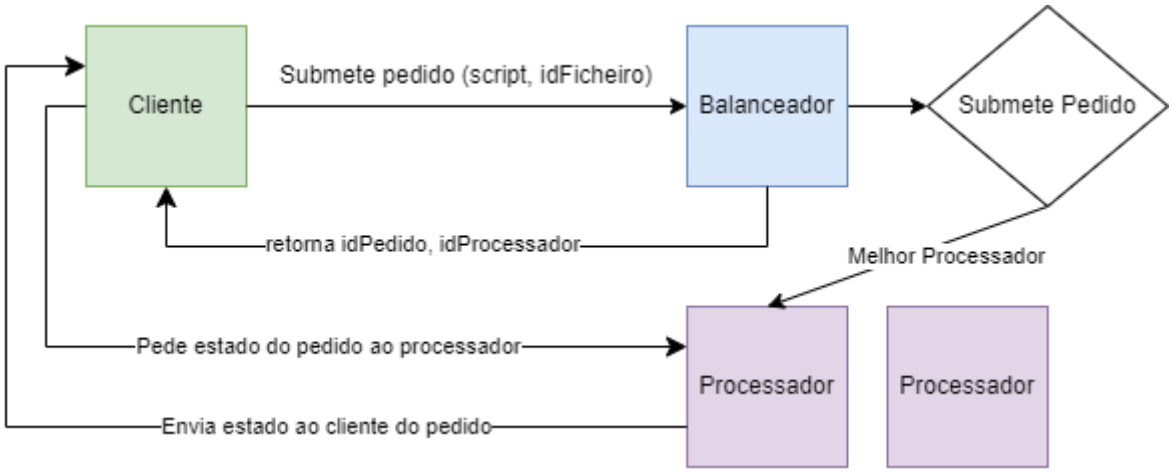


Figura 3 : Diagrama - Requisito Funcional 2 e 3

4.1.1. Resolução do Requisito Funcional 2

Criou-se dois projetos, o Balanceador e o Processador. O Balanceador distribui a carga pelos Processadores, e o Processador executa o script sobre o ficheiro, guardando o resultado na camada de armazenamento.

O Cliente submete ao Balanceador um pedido com o caminho absoluto para o script e o ID do ficheiro.

Assim que o Balanceador recebe um pedido, envia os dados ao melhor processador, e este por sua vez é o que tem menos pedidos. Assim que o Processador recebe o pedido, começa a executar e retorna o ID do pedido para o Balanceador.

O Balanceador irá retornar ao cliente, o ID do pedido e o URL do processador ao qual se submeteu o pedido do Cliente.

```java
HugoNeves0808 +2
@Override
public List<Object> submetepedido(String filePath, UUID ficheiro) throws MalformedURLException, NotBoundException, RemoteException {
    String processador = getBestProcessador();
    //String pUrl = processadores.get(processador);
    ProcessorInterface processor = (ProcessorInterface) Naming.lookup(processador);
    //submete e recebe o uuid do pedido
    UUID pedidoId = null;
    try {
        pedidoId = processor.submetePedido(filePath,ficheiro);
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }

    return Arrays.asList(processador, pedidoId);
}
```

Figura 4 : Função - Envio do pedido ao melhor processador

```java
maraines90 +3 *
@Override
public UUID submetePedido(String path, UUID ficheiro) throws RemoteException, MalformedURLException, NotBoundException {

    Pedido p = new Pedido(path,ficheiro,UUID.randomUUID());
    estadoPedido.put(p.getPedidoId(),p);
    ReplicaManagerInterface r = (ReplicaManagerInterface) Naming.lookup( name: "rmi://localhost:2030/replicaManager");
    r.addPedido( urlProcessor: "rmi://localhost:"+port+"/processor",p);
    p.start();
    return p.getPedidoId();
}
```

Figura 5 : Função - Iniciar o pedido numa nova *thread*

4.1.2. Resolução do Requisito Funcional 3

Quando o Cliente recebe a informação do pedido, o ID do pedido e o URL do processador, o Cliente vai pedir o estado do pedido ao Processador. Enquanto o estado for "*waiting*", o Cliente repete o pedido a cada segundo.

```java
String status = processor.getEstado(pedidoId);
while(!Objects.equals(status,  b: "Done")) {
    System.out.println("Estado do pedido " + pedidoId + " no servidor " + urlProcessador + " é " + status);
    status = processor.getEstado(pedidoId);
    Thread.sleep( millis: 1000);
}
```

Figura 6 : Função - Cliente pede o estado do pedido ao Processador

```java
public String getEstado(UUID idPedido) throws RemoteException{
    if(!estadoPedido.containsKey(idPedido)) {//se o pedido não existir
        System.out.println("O processo " + idPedido + " não existe");
        return "Z";
    }
    return estadoPedido.get(idPedido).getStatus();
}
```

Figura 7 : Função - Processador devolve o estado do pedido ao Cliente

5. Sprint 3

5.1. Descrição do Sprint 3

rf4. O cliente obtém o resultado do processamento junto do *storage*, utilizando o identificador do pedido.

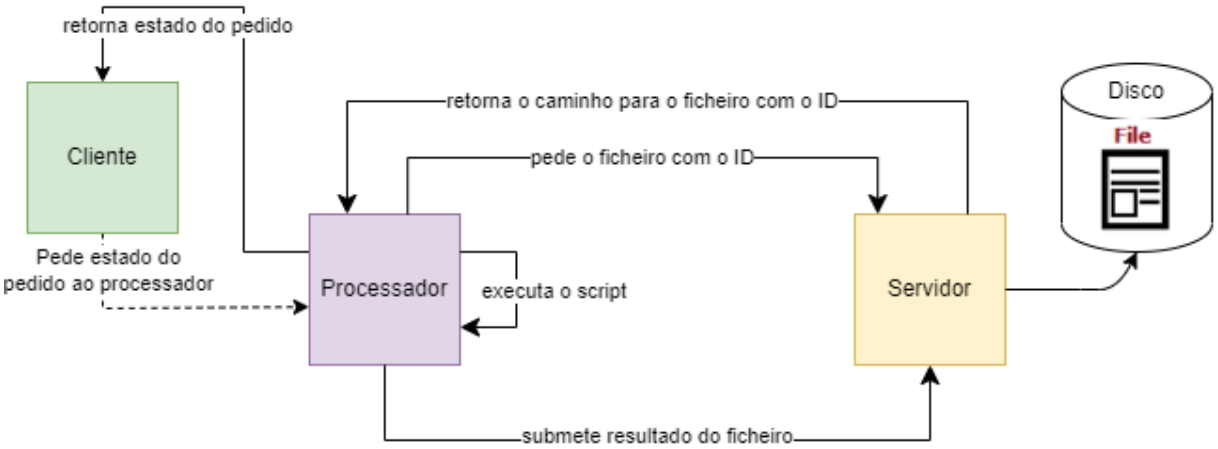


Figura 8 : Diagrama - Requisito Funcional 4

5.1.1. Resolução do Requisito Funcional 4

Quando o Cliente pede o estado de um pedido ao Processador, o mesmo é colocado numa lista de espera, onde posteriormente vão ser executados por ordem de chegada. Porém, antes de serem executados, o Processador pede o ficheiro com um ID ao Servidor. Seguidamente o Servidor retorna para o Processador o caminho para o ficheiro com o ID. O Processador executa o script, e envia o resultado.

Os pedidos têm vários estados, antes de um pedido ser executado, o mesmo está no estado "*waiting*".

Quando um pedido está a ser executado, o estado é alterado para "*running*".

Quando o processador já executou o script e a envia o resultado do ficheiro para o Servidor, o estado é alterado para "*writing*".

Por fim, quando a execução acabar, o estado do pedido é alterado para "*done*".

6. Sprint 4

6.1. Descrição do Sprint 4

rnf1. Os processadores devem fazer o *broadcast* de *heartbeats* periódicos com a informação dos seus recursos e tarefas. Desta forma, o balanceador obtém informação dos recursos disponíveis em cada processador e o coordenador determina quais são os processadores válidos (sem falhas).

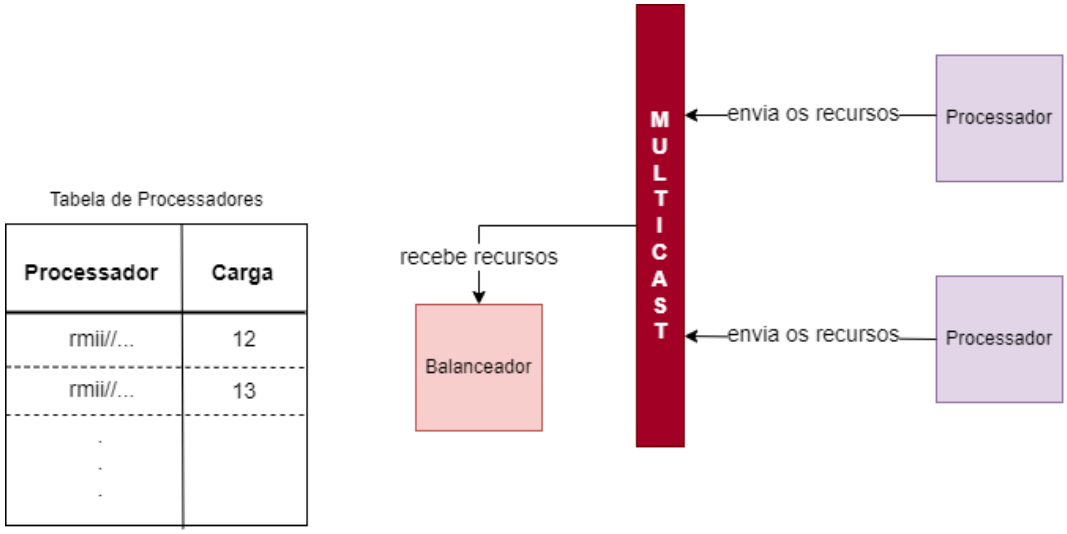


Figura 9 : Diagrama - Requisito Não Funcional 1

6.1.1. Resolução do Requisito Não Funcional 1

Os Processadores vão enviar *heartbeats* periódicos com a informação dos seus recursos ao Balanceador.

Quando o Balanceador recebe um *heartbeat* de um Processador, vai atualizar os seus recursos, caso ainda não exista na tabela interna, adiciona-o.

```
26              String[] parts = received.split( regex: ";");
27              String tipo = parts[0];
28              String address = parts[1];
29
30              if(Objects.equals(tipo, b: "setup")){
31                  b.addProcessador(address, pedidos: 0);
32              }else if(tipo=="update"){//update
33                  String nPedidosWating = parts[2];
34                  b.updateProcessor(address,Integer.parseInt(nPedidosWating));
35                  System.out.println("No processador "+address+" há "+nPedidosWating+ " pedidos à espera");
36              }
37          }
```

Figura 10 : Função - Receção/interpretação dos *heartbeats* dos Processadores

7. Sprint 5 e 6

7.1. Descrição e resolução do Sprint 5 e 6

(Sprint6) rnf2. Quando o balanceador arranca, deverá contactar o coordenador para obter a lista de processadores válidos. Nesta fase, o balanceador não conhece o endereço do coordenador atual.

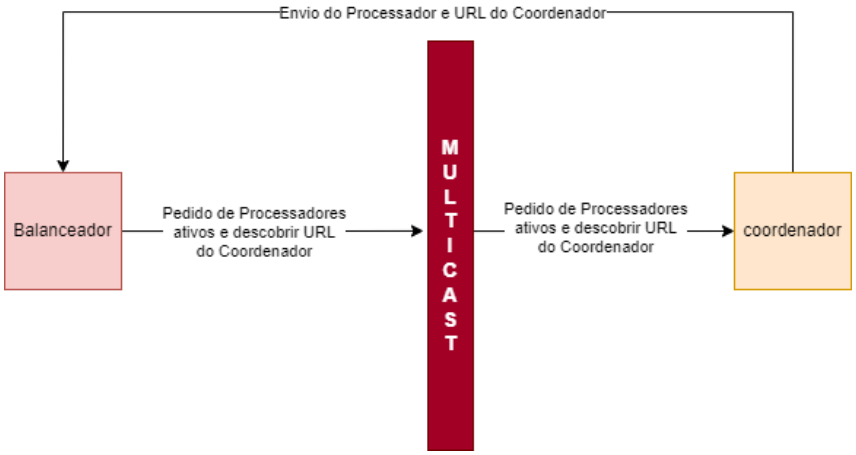


Figura 11 : Diagrama - Requisito Não Funcional 2

(Sprint5) rnf3. Quando um coordenador deixa de receber *heartbeats* de um processador durante um período de tempo superior a 30 seg, deverá remover o respetivo processador da lista de processadores disponíveis. Posteriormente, notifica o processador com mais recursos, para resumir os pedidos pendentes do processador com falhas, e o balanceador para comunicar a falha de um dos processadores.

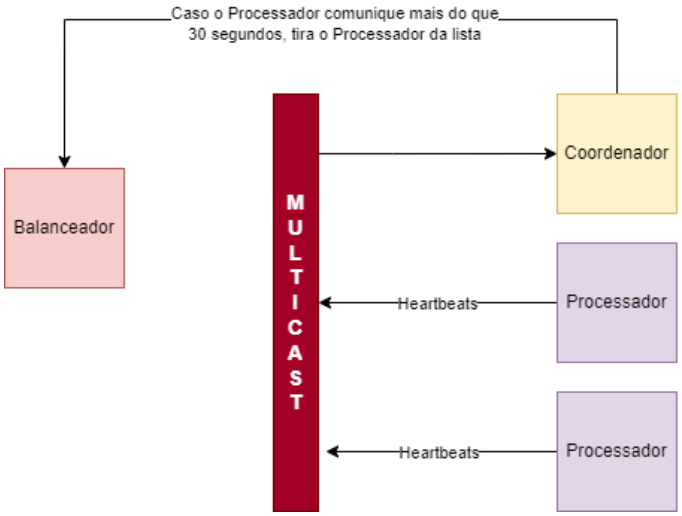


Figura 12 : Diagrama - Requisito Não Funcional 3

(Sprint5) rnf4. Quando um coordenador recebe um *heartbeat* do tipo "*setup*", vindo de um processador que não se encontra na lista, adiciona o processador e notifica o balanceador.

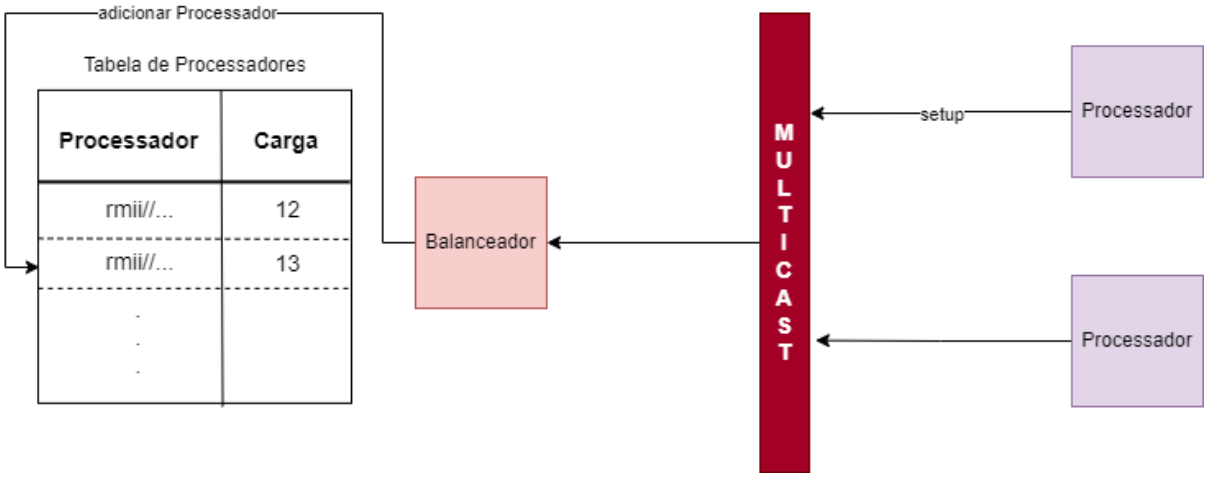


Figura 13 : Diagrama - Requisito Não Funcional 4

(Sprint6) rnf5. Os ids dos elementos e dos pedidos devem ser universais

Como podemos ver pela figura 14, podemos observar tanto os ficheiros, como os pedidos por um UUID (*Universally Unique Identifier*).

```
@Override
public UUID submetePedido(String path, UUID ficheiro) throws RemoteException, MalformedURLException, NotBoundException {
    Pedido p;
    if(storageLeaderUrl == null) {
         p= new Pedido(path, ficheiro, UUID.randomUUID(), backUp, url);
    }else{
        p = new Pedido(path, ficheiro, UUID.randomUUID(), backUp, url,storageLeaderUrl);
    }
    estadoPedido.put(p.getPedidoId(),p);

    p.start();
    return p.getPedidoId();
}
```

Figura 14 : Função - Ficheiros e pedidos por um UUID

(Sprint6) rnf6. O acesso aos recursos partilhados por várias *threads* têm de ser *thread-safe*

Como podemos observar na figura 15, temos exemplo de variáveis *thread-safe*. Neste caso é do coordenador.

Figura 15 : Função - Exemplo de variáveis thread-safe

(Sprint6) rnf7. A comunicação *unicast* deverá ser realizada através do *middleware* RMI

Como exemplo deste requisito não funcional, apresentamos a comunicação do Cliente com o Servidor de dados, para enviar um ficheiro, tal como mostra a figura 16.

```
//Sprint 1
Registry storage = LocateRegistry.getRegistry( host: "localhost",  port: 2022);
FileManagerInterface fileServer = (FileManagerInterface)storage.lookup( name: "filelist");

File objF = new File(fileName);
// Creating an OutputStream
byte [] mydata=new byte[(int) objF.length()];
FileInputStream in=new FileInputStream(objF);

in.read(mydata,  off: 0, mydata.length);
in.close();

System.out.println("uploading to server...");
fileUuid = fileServer.uploadFileToServer(mydata, (int) objF.length());
```

Figura 16 : Função - Comunicação Cliente com o Servidor de dados

(Sprint6) rnf8. A execução dos pedidos deverá ser FIFO *ordering*

(Sprint6) rnf9. Os processos da camada de *storage* garantem consistência eventual, onde o líder é eleito por um algoritmo de consenso

Neste requisito não funcional 9, foi implementado um algoritmo *raft* para garantir consistência eventual na camada *storage*. Na figura 17 está a parte da decisão de um nó se tornar candidato a líder.

```
private void startElectionTimeout() {
    dany9161
    final Runnable setupElectionRunner = new Runnable() {
        dany9161
        public void run() {
            nVotos=1;
            term++;
            MulticastPublisher mp = new MulticastPublisher();
            try {
                mp.sendMulticastMessage("e;" + term + ";" + nOperations + ";" + url);

            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        }
    };
    elctionTimeout.schedule(setupElectionRunner, (int)Math.floor(Math.random()*(3000-1000+1)+3000), TimeUnit.MILLISECONDS);
}

1 usage  dany9161
private void stopElectionTimeout(){
    elctionTimeout.shutdownNow();
}

1 usage  dany9161
private void resetElectionTimeout(){
    stopElectionTimeout();
    startElectionTimeout();
}
```

Figura 17 : Função - Nó candidato a líder

(Sprint6) rnf10. A comunicação *multicast* deverá garantir a integridade

A palavra-chave *synchronized* no Java não permite que um método seja executado em simultâneo, tal como mostra a figura 18.

```
synchronized public void sendMulticastMessage(String multicastMessage) throws IOException {
    socket = new DatagramSocket();
    group = InetAddress.getByName( host: "230.0.0.1");
    buf = multicastMessage.getBytes();

    DatagramPacket packet = new DatagramPacket(buf, buf.length, group,  port: 4446);
    socket.send(packet);
    socket.close();
}
```

Figura 18 : Função – *Synchronized*

(Sprint6) rnf11. A falha do coordenador pode ser detetada por qualquer processador, mas só é considerada definitiva por consenso por maioria

Na figura 19, podemos ver a rotina para controlar se o Coordenador está ativo.

```
Runnable checkCoordenatorRunnable = new Runnable() {
    dany9161 +1
    public void run() {
        if ((System.currentTimeMillis()- lastCHeartBeat.getTime()) > 30000 && cActive){//se o ultimo heartbeat recebido do coordenador
            //enviar a mensagem que o controlador morreu
            MulticastPublisher mp = new MulticastPublisher();
            try {
                mp.sendControlerDeadMessage( multicastMessage: "m;"+url);
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
            avaliaCMorreu();
        }
    }
};

ScheduledExecutorService checkCoordenatorExecutor = Executors.newScheduledThreadPool( corePoolSize: 1);
checkCoordenatorExecutor.scheduleAtFixedRate(checkCoordenatorRunnable,  initialDelay: 0,  period: 3, TimeUnit.SECONDS);
}
```

Figura 19 : Função - Rotina para controlar se o Coordenador está ativo

Quando o coordenador envia uma mensagem, assinala que está ativo.

```
public void refreshControlador(int part) {
    if (!cActive) System.out.println("O controlador voltou");
    cActive=true;
    lastCHeartBeat=new Time(System.currentTimeMillis());
    nPAtivos = part;
    nPCDead =0;
}
```

Figura 20 : Função - Coordenador envia uma mensagem

(Sprint6) rnf12. O sistema distribuído apenas considerar falhas do tipo fail-stop

---

Neste projeto existem 3 tipos *heartbeats*, que são:

- *Heartbeat coord*
- *Heartbeat Processs*
- "*Heartbeat" Storage Leader*

Mensagens *Multicast*:

- C *heartbeat* - h;nP
- P confirmar que o C morreu - m;urlP

- P heartbeat - setup;urlProcessor
- P setup - update;urlProcessor;
- B pedir processadore ativos - b;urlBalanceador
- S comando - c;termo;nOperation;fileId;bytes[];urlLeader
- S eleicao - e;termo;nOperation;urlCandidate

8. Melhoramentos

Como é costume, em todos os projetos existem sempre melhorias a ser feitas, e este não foge a essa realidade.

Deixamos aqui algumas sugestões de melhoria:

- A replica dos Processadores ao guardar todos os pedidos, e não só os que estão por fazer;
- O Processador ao receber a lista de pedidos do Processador que falhou, junta os feitos à sua lista e só executa os que não foram pedidos, porém, no nosso projeto, o Processador só recebe e executa os que não foram terminados;
- Se o Cliente não conseguir contactar o Processador responsável por um pedido, perguntar ao Balanceador qual é o Processador responsável por esse pedido. O Balanceador deve enviar um *multicast* a perguntar quem têm esse pedido, o Processador responsável dá sinal, e o Balanceador informa o Cliente.
- Criar *backUp*/redundância no Coordenador, para o caso deste falhar;
- Ser o Processador a atualizar o seu *backUp*, sendo que neste momento é o pedido a atualizar;
- Criar *backUp*/redundância no Balanceador.
- Alterar a métrica para contabilizar os pedidos que estão a espera de ser executados e os que estão a ser executados.

9. Conclusão

Ao longo do projeto deparamo-nos com várias dificuldades, como a implementação de replicas dos processadores, a implementação do algoritmo *raft* na camada storage, etc. Conseguimos ultrapassar estes desafios com a ajuda do material disponibilizado pelo Professor e de artigos encontrados na internet.

Com a realização deste trabalho percebemos que o planeamento e o desenvolvimento dos sistemas distribuídos é difícil, vistos serem processos complexos, tendo de ter atenção a vários aspetos para que fique uma aplicação robusta.

De forma geral, concluímos que a importância dos Sistemas Distribuídos para um programa é essencial, visto atuarem diretamente nas funcionalidades que vão permitir a escalabilidade.

A realização deste projeto foi uma mais-valia para todos os elementos do grupo, por isso, encontramo-nos gratos e melhor preparados para o futuro.

10. Bibliografia

arcgis. (s.d.). *arcgis*. Fonte: arcgis: https://help.arcgis.com/en/sdk/10.0/java_ao_adf/api/arcobjects/com/esri/arcgis/geodatabase/Replica.html

RuchitaGarde. (s.d.). *Github*. Fonte: RuchitaGarde GitHub: https://github.com/RuchitaGarde/RMI

Zhang, Z. (2021, Dezembro 8). *Raft Algorithm, Explained*. Retrieved from Medium: https://towardsdatascience.com/raft-algorithm-explained-a7c856529f40