Googol

Motor de Pesquisa

1ª Meta

Da autoria de:
Eduardo Figueiredo 2020213717
Fábio Santos 2020212310

Índice

Introdução

Este projeto tem como objetivo por em prática os conhecimentos lecionados na cadeira de Sistemas Distribuídos, criando de um motor de pesquisa semelhante ao Google ou Bing. Ao longo do desenvolvimento do projeto foram usadas técnicas como sincronização, comunicação através de Java RMI, comunicação através de Sockets (mais concretamente Multicast), Threads, sinais, Base de Dados, entre outros.

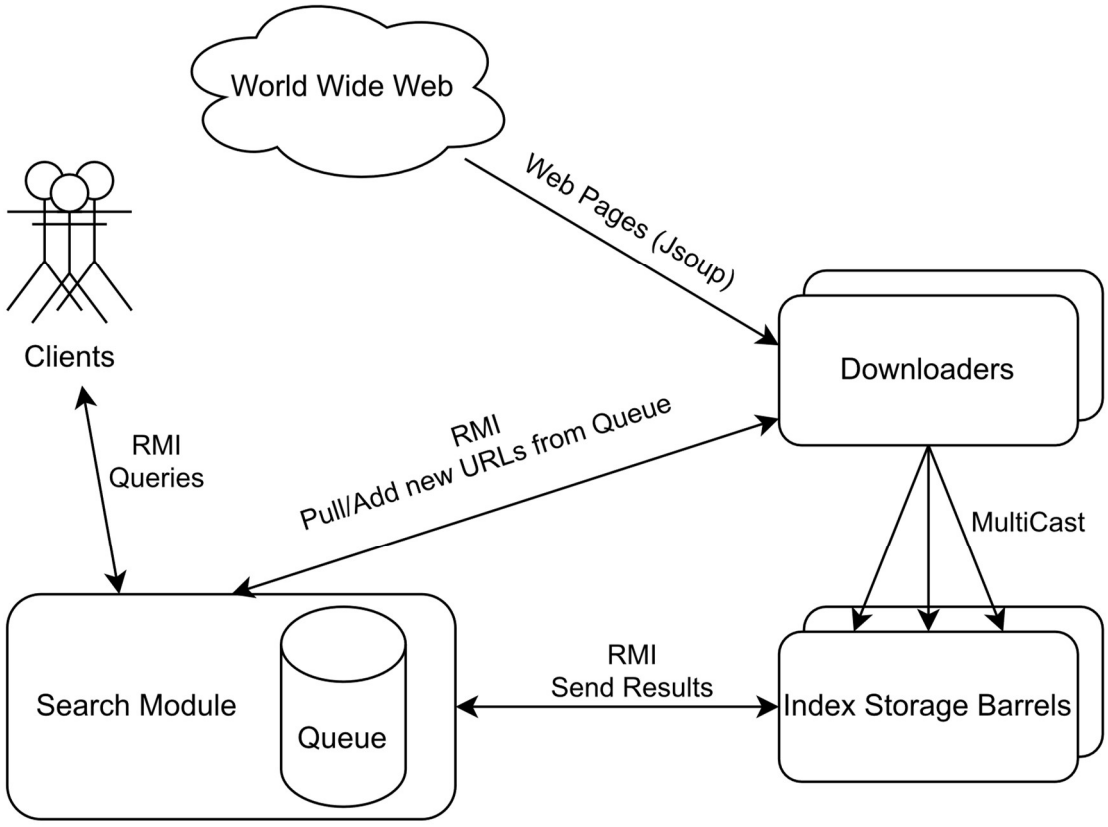
Para isso, a aplicação é composta por programas que trabalham em simultâneo possibilitando aos utilizadores funcionalidades como:

- Registar-se ou Logar-se na plataforma
- Indexar um novo URL
- Pesquisar por páginas que contenham um conjunto de termos
- Pesquisar que URLs contém uma ligação para um dado URL
- Ver as estatísticas da plataforma
    - Quantos programas estão a correr ao mesmo tempo
    - As 10 pesquisas mais comuns

A aplicação foi implementada de forma a conseguir recuperar de eventuais crashes no Sistema (failover). Quaisquer dados que forem gerados ou alterados na sessão que porventura crashou serão guardados em ficheiros que futuramente, após a recuperação do sistema ou no próximo arranque do mesmo, serão lidos para recuperar estes dados.

Arquitetura de Software

Como dito anteriormente, a aplicação desenvolvida é composta por vários programas que trabalham em simultâneo. Estes são o **Search Module**, o(s) **Downloader**(s), o(s) **Index Storage Barrel**(s) e a aplicação usada pelo(s) **Cliente**(s). É possível que haja várias instâncias da mesma aplicação a correr ao mesmo tempo para tornar o Motor de pesquisa mais rápido e eficiente. Optámos por preferir ter vários programas **Downloader** ou **Barrel** a correr em simultâneo em vez de um programa *Multithreaded* que contivesse vários destes objetos para aproximar o nosso projeto de um Sistema Distribuído real. Desta maneira seria possível ter vários destes programas a correr em máquinas diferentes, distribuindo o processamento por elas.



Search Module

O Search Module é a porta de entrada para o Sistema. É ele quem faz a ponte entre os vários programas que constituem a aplicação. O Search Module comunica com o(s) **Downloader**(s), o(s) **Index Storage Barrel**(s) e com o(s) **Cliente**(s) através de Java RMI, além de gerir a Base de Dados.

Para o Motor de Busca poder funcionar é necessário que pelo menos um URL tenha sido indexado, isto é, um cliente não consegue tirar partido da aplicação se esta não conhecer nenhum URL. Assim, sempre que um cliente quiser indexar um URL, este URL é colocado numa **Dequeue** para mais tarde ser analisado por um **Downloader**. Esta **Dequeue** está contida no Search Module e não é um programa independente.

Sempre que um cliente faz um pedido, como por exemplo a busca de uma palavra ou a visualização das estatísticas do Sistema, o **Search Module** escolhe de forma circular um **Index Storage Barrel** para processar o seu pedido.

O **Search Module** implementa a *GoogleInterface.java* para responder às pesquisas dos **Clientes**, a *StorageBarrelInterface.java* subscrever os **Barrels**, isto é, saber quantos e quais **Barrels** estão em funcionamento no momento, e a *DownloaderInterface.java* para subscrever os **Downloaders**. São usadas então duas Interfaces em cada uma das comunicações de forma a usar os *call-backs* necessários: uma para subscrever cada um dos outros programas usando uma lista de interfaces, outra para os pedidos efetivos entre os vários programas. O conhecimento de quantos **Downloader** e **Barrels** estão a trabalhar é necessário para distribui o trabalho entre os vários **Downloaders** e **Barrels** e, por outro lado, para poder ser mostrado na página de administração em tempo real.

```
1 public class RMISearchModule extends UnicastRemoteObject implements
   GoogolInterface, StorageBarrelInterface, DownloaderInterface {
2     static ArrayList<StorageBarrelInterfaceB> listOfBarrels;
3     static ArrayList<DownloaderInterfaceC> listOfDownloaders;
```

Dequeue

A Queue implementada na verdade é uma **Dequeue**. É um objeto contido dentro do Search Module, implementa uma *Linked Blocking Dequeue*, disponibilizada na biblioteca *java.util.concurrent*. Esta implementação de uma **Dequeue** é bloqueante, isto é, existe um semáforo que bloqueia a **Dequeue** sempre que esta está a ser utilizada por outro processo (threadsafe) o que previne que vários **Downloaders** retirem e processem o mesmo URL. Para além deste semáforo existe um outro que bloqueia a **Queue** sempre que esta esteja vazia, de forma a não haver esperas ativas neste caso.

Quando um utilizador tenta indexar um URL é normal que a **Dequeue** já tenha vários URLs em espera para serem indexados. De forma a um utilizador poder fazer uma pesquisa sobre o URL que acabou de indexar, o seu URL é colocado na cabeça da **Dequeue** para ter prioridade sobre os outros URLs. Este

objeto contém métodos onRecovery() e onCrash() que são usados para guardar ou recuperar o estado da Dequeue no arranque e num crash do sistema, respetivamente.

```java
public void onCrash() {
    System.out.println("Queue: System crashed, saving URL queue state.");
    if (queue.size() != 0) {
        try (FileOutputStream fos = new FileOutputStream(file);
                ObjectOutputStream oos = new ObjectOutputStream(fos)) {
            oos.writeObject(queue);
            oos.close();
            fos.close();
        } catch (IOException ioe) {
            System.out.println("Error trying to write to \"QUEUE.obj\".");
        }
    }
}
```

Downloader

O **Downloader** é o programa que trata de fazer o crawl do URL. Pode existir mais que um **Downloader** a trabalhar ao mesmo tempo. Se for este o caso, mais URLs serão processados por unidade de tempo. Quando o programa começa, existe uma comunicação entre o **Downloader** e o **Search Module** de forma ao **Search Module** saber que existe (mais) um **Downloader** a trabalhar (subscribe), adicionando-o à lista de **Downloaders** ativos. De forma semelhante, se um **Downloader** encerrar ou crashar o **Search Module** é notificado, eliminando-o da lista (unsubscribe). Esta comunicação é possível graças à interface *DownloaderInterface.java* usada no **Search Module**.

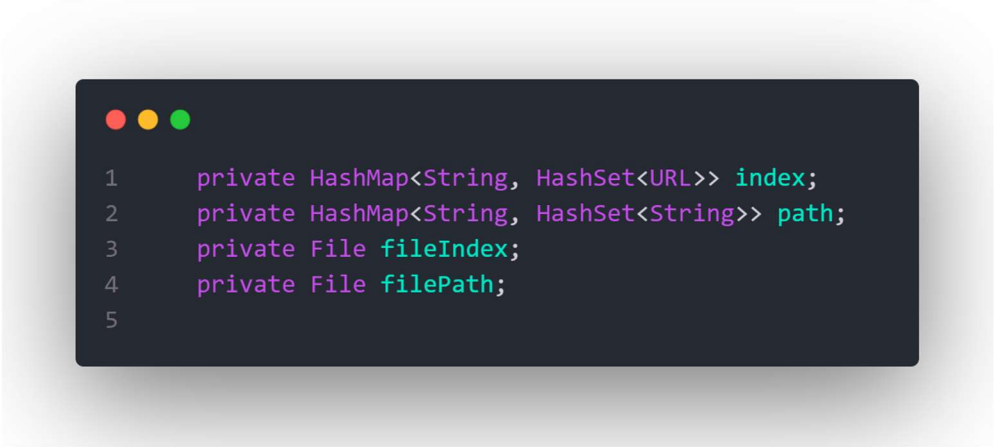
Com a ajuda da biblioteca *JSOUP*, o ficheiro HTML obtido é processado, distinguindo as palavras que o constituem, do título, de outros URLs presentes. Estes URLs serão adicionados à **Dequeue** para serem processados futuramente, desta vez na cauda da mesma. Para manter o Sistema mais eficiente, verifica-se se cada uma das palavras que constituem o HTML é uma *Stop Word*.

De forma a guardar a informação obtida pelo crawl efetuado, o **Downloader** envia por *Multicast* para os **Index Storage Barrels** guardarem a informação obtida.

Index Storage Barrel

O **Index Storage Barrel** é o programa que guarda toda a informação gerada pelo Sistema. É comum que haja mais que um **Barrel** a trabalhar ao mesmo tempo para haver redundância de informação e assim mais facilmente recuperar de erros. Para isso, os Barrels guardam informação em ficheiros de objetos e possuem 2 estruturas de dados:

- Index: Hashmap constituído por uma String de uma palavra e por um Hashset de URLs que contém essa palavra. Muitas vezes o Index é chamado de Índice Invertido.
- Path: Hashmap constituído por uma String de um URL[1] e por um Hashset de URLs que têm uma ligação para URL[1].



Um **Index Storage Barrel** por si só é um programa *MultiThreaded*, uma vez que precisa de guardar toda a informação obtida pelos **Downloaders** recebida por *Multicast* (Thread1) bem como responder pedidos dos **Clientes** (Thread2).

O **Index Storage Barrel** implementa a *StorageBarrelInterfaceB.java* para enviar para o **Search Module** os resultados das pesquisas por palavras ou por URL que um **Cliente** fez.

Mais uma vez, existem mecanismos de deteção de erros que guardam o Index e o Path caso haja algum crash, recuperando deles de seguida.

Sincronização

Por vezes acontece que um **Barrel** inicia primeiro que outro o que torna a informação diferente nos dois **Barrels**. Para contornar esta situação é feita uma sincronização entre os **Barrels**. Através de semáforos, o **Search Module** lê a informação presente em todos os **Barrels** (Index e Path) e compara-a. Se alguma deles for nula, é feita uma copia para o **Barrel** que acabou de iniciar, caso ambos tenham informação é feito *merge* da mesma para ambos os **Barrels**.

Cliente

O **Cliente** quando inicia a aplicação tem a opção de se registar no Googol ou de fazer login. Os seus dados são encriptados e enviados para o **Search Module** para este verificar se os dados estão de acordo com a **Base de Dados**.

O **Cliente** invoca métodos remotos para obter os resultados das suas pesquisas, tirando partido da interface *GoogleInterface.java*. Para isso, é necessário que o **Search Module** esteja ativo e haja pelo menos um **Barrel** a funcionar.

```
Welcome to Googol

1 - Login
2 - Register
1
Enter your username:
fabio
Enter your password:
teste

Hi, fabio

1 - Index URL
2 - Pages with word
3 - Pages with URL
4 - Show Stats
0 - Exit

1
Type the URL you want to Index:
https://www.uc.pt
```

Funcionalidades implementadas

1 Indexar um novo URL

O **Cliente**, após fazer o login, consegue enviar um URL para ser indexado. A string inserida no terminal é enviada para o **Search Module** por RMI, que por sua vez é colocada na cabeça da **Dequeue**. Um **Downloader** disponível, retira-o da **Dequeue** para ser analisado. Os resultados da pesquisa são colocados num objeto URL que contém o título da página, todas as palavras encontradas (exceto StopWords em português e inglês), uma citação (primeiras 20 palavras do site) e todos os URLs contidos na página. Este objeto é então enviado por Multicast para todos os **Barrels** disponíveis. Os **Barrels** pegam nesta informação e guardam-na nas respetivas estruturas de dados. O índice invertido é criado pegando numa palavra presente num URL ligando-a à lista de URL que contém essa palavra. O Path é criado de forma semelhante, usando os URLs contidos num dado URL. Esta informação é guardada em cada um dos Barrels e mais tarde no ficheiro de objetos correspondente.

```
// Search Module
public void newURL(String URLString) throws RemoteException {
    System.out.println("Search Module: Adding \"" + URLString + "\" to the QUEUE");
    urlQueue.addURLHead(new URL(URLString));
}
```

```
System.out.println("Downloader: System started");
while (true) {
    try {
        // Pega o ultimo URL da Fila e faz o crawl
        URL url = SMi.getURLQueue();
        System.out.println("Downloader: Indexing " + url);
        url = downloader.crawlURL(url, SMi);
```

2 Indexar Recursivamente um URL encontrado

Durante a indexação de um URL é possível encontrar ligações para outros URLs. Para criar o índice invertido de forma recursiva, os novos URLs forem encontrados nesta análise serão colocados na cauda **Dequeue**, de forma a dar prioridade a URLs inseridos na cabeça pelos utilizadores. Estes novos URLs encontrados passam por todo o processo da primeira funcionalidade.

3 Pesquisar páginas que contém um conjunto de termos

O **Cliente** pode naturalmente fazer uma pesquisa no motor de buscar composta por um conjunto de termos. Para isso, os termos digitados por ele no terminal viajam por RMI para o **Search Module** distribuir o processamento da pesquisa. Inicialmente, as palavras pesquisadas são inseridas numa tabela da base de dados que contém a palavra e o número de vezes que foi pesquisada. Estes dados serão usados quando um cliente acede à página de administração do Sistema.

O **Search Module** escolhe então um **Barrel** ativo para fazer o pedido, visto que os **Barrels** são quem possuem a toda a informação do Sistema. Os **Barrels** percorrem então o Index à procura das palavras inseridas pelo utilizador e devolvem um conjunto de URLs que contém todas as essas palavras. No lado do **Cliente**, os resultados da sua pesquisa são apresentados em páginas de 10 URLs, sendo-lhe possível avançar e recuar nas mesmas.

```
1  // Choose a barrel to work (circular)
2  StorageBarrelInterfaceB Barrel = listOfBarrels.get((nextBarrel++) %
   listOfBarrels.size());
3  HashSet<URL> hash = Barrel.getUrlsToClient(words, pages);
4  ret = "\n";
5  if (hash != null) {
6      for (URL url : hash) {
7          ret += url.toString() + '\n';
8      }
9      return ret;
10 } else if (hash == null && pages > 0) {
11     return "\nThere are no more Urls with that word!";
12 } else {
13     return "\nThere are no Urls with that word!";
14 }
```

4 Resultados ordenados por importância

Os resultados apresentados nas pesquisas da funcionalidade anterior são apresentados por ordem de importância. Um URL considera-se mais importante que outro se possuir mais ligações para ele. Assim, para ordenar os resultados da pesquisa, cruzam-se os dados do Index com os dados do Path: procura-se a palavra inserida pelo utilizador no índice invertido, obtêm-se os URLs que a contém e pesquisa-se no Path a quantidade de ligações que levam até esse URL, ordenando a pesquisa segundo este tamanho.

Primeiramente, é criado um Hashset (commonValues) de URLs que apenas possui os URLs que contém uma das palavras dadas como *key* do Index. A partir dele é criado um *ArrayList* de Objetos Relevance, classe esta que contém um URL e a sua relevância. Esta lista é então ordenada de forma inversa através de um *Comparator* presente em *java.util.Colletions*.

```
// Order the results by relevance
ArrayList<Relevance> ordered = new ArrayList<>();
for(URL url: commonValues){
    Relevance aux = new Relevance(url, path.get(url.getUrl()).size());
    ordered.add(aux);
}
// reverse order
Collections.sort(ordered, new Comparator<Relevance>() {
    @Override
    public int compare(Relevance r1, Relevance r2) {
        return r2.getRelevance() - r1.getRelevance();
    }
});
```

Após um **Cliente** efetuar a sua busca, se existirem mais do que 10 resultados, estes serão divididos em várias páginas com 10 URLs cada uma. O **Cliente** depara-se com um menu onde é possível avançar ou recuar das páginas da pesquisa e sair da própria. Para isso, a função invocada remotamente pelo Cliente possui um contador de páginas como argumento que aumenta o diminui conforme o utilizador avança ou recua na pesquisa.

5 Consultar lista de páginas com ligações para uma página especifica

É possível também consultar a lista de URLs que contenham um link para outro URL. De forma semelhante à funcionalidade 3, o Cliente pode inserir um URL e ver a lista de URLs que levam até ele. Esta informação é dada pelo Path de um Barrel escolhido pelo Search Module.

```
public HashSet<String> getpagesWithURL(String URL, int pages)
throws RemoteException {
    // Uses pagesWithULR
    System.out.println("Barrel: Sending URLs that lead to " + URL);
    if (path.containsKey(URL)) {
        return path.get(URL);
    } else {
        return null;
    }
}
```

6 Página de Administração em tempo real

Qualquer **Cliente** registado tem a possibilidade de ver as estatísticas do Sistema em tempo real. Estas estatísticas são:

- O número de **Downloaders** ativos e o identificador de cada um
- O número de **Barrels** ativos e o identificador de cada um
- As 10 pesquisas mais comuns, constituída pela palavra e o número de vezes que foi pesquisada

Esta informação é enviada para os **Clientes** via RMI, dado que é esta a comunicação existente entre eles e o **Search Module**. O número de **Downloader** e **Barrels** é dado pela lista de cada uma destas interfaces presente no Search Module enquanto as pesquisas mais comuns são obtidas por uma *Query* à Base de Dados, que foi previamente preenchida à medida que os **Clientes** faziam as suas pesquisas, como dito anteriormente.

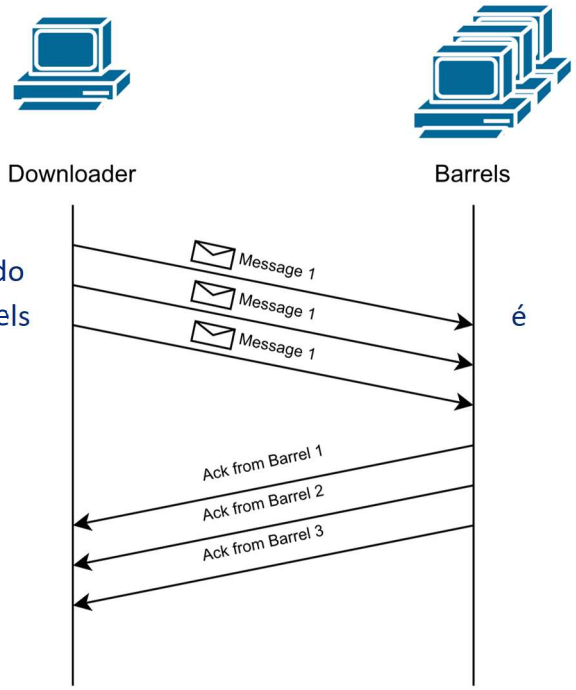
```
String check = "SELECT word, num FROM topSearches ORDER BY num DESC";
PreparedStatement checkStatement = connection.prepareStatement(check);
ResultSet rs = checkStatement.executeQuery();
String word = "";
int num = 0;

while (rs.next()) {
    word = rs.getString("word");
    num = rs.getInt("num");
    result += word + ": " + num + "\n";

    if (count++ == 10)
        break;
```

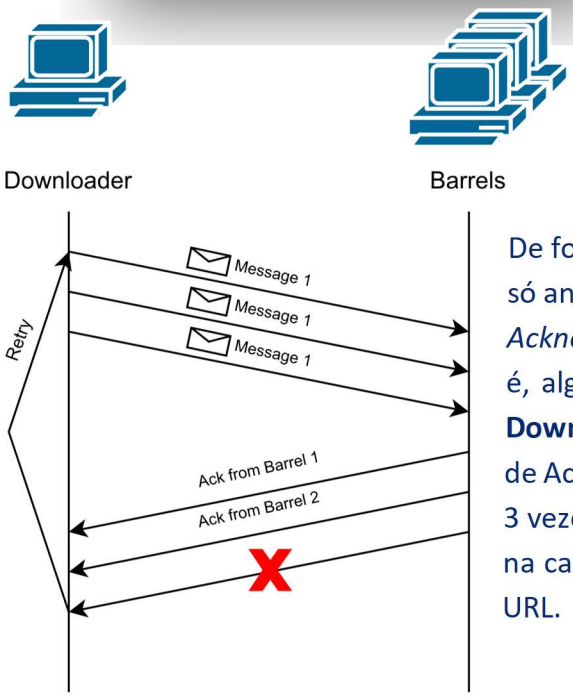
Protocolo UDP Multicast

O protocolo Multicast implementado consiste em enviar um objeto Message para todos os **Barrels** ativos. Esta classe contém o objeto URL a ser guardado e a porta UDP usada para os **Barrels** comunicarem de novo com o **Downloader** em questão. Do lado do **Downloader** é enviado apenas o tamanho da Message enquanto do lado dos Barrels recebido o tamanho máximo do pacote.




```java
public class Message implements Serializable{

    private URL url;
    private boolean zip;
    private int lenght;
    private int PORT;
```



De forma a tornar o protocolo Multicast fiável, os **Downloaders** só analisam o próximo URL quando todos os **Barrels** enviarem o *Acknowledgment.* Se por alguma razão existir um timeout, isto é, algum pacote tiver sido perdido durante a comunicação, o **Downloader** repete o processo de envio de Message e recessão de Ack no máximo 3 vezes. Caso o aconteça timeout mais do que 3 vezes, o **Downloader** envia o URL para a **Dequeue** novamente na cauda da queue e assim prossegue para o crawl do próximo URL.

Uma vez havendo timeout, o **Downloader** informa o **Search Module** de tal situação. O **Search Module** faz então um ping para todos os Barrels de forma a perceber se estão ativos ou houve algum tipo de crash. Se algum deles não responder, esse **Barrel** é retirado da lista de **Barrels** ativos contida no **Search Module**, prevenindo que mais timeouts aconteçam por causa do Sistema pensar que o **Barrel** em questão está em funcionamento.

Requisitos não-funcionais

Tratamento de Exceções e Failover

A aplicação deve estar disponível para o utilizador em qualquer altura, quer tenha acontecido algum erro ou não. Para isso, foram implementados os métodos necessários para recuperar de erros. Programas como o **Search Module**, o **Index Storage Barrel** e a **Dequeue** têm métodos de deteção de crashes métodos que recuperam destes eventuais crashes. Estas funções escrevem e leem de ficheiros de objetos sempre que são executadas.

Qualquer um destes programas, quando inicia, usa o método onRecovery() para ler os dados da ultima utilização da aplicação. Assim, nenhuns dados são perdidos no final de cada sessão, isto porque a aplicação pode encerrar e quando se iniciar futuramente é apropriado que se tenha os dados obtidos da última sessão.

Os métodos onCrash() são usados sempre que alguma exceção ocorra, como por exemplo quando os **Barrels** não conseguem ler do socket usado no Multicast para guardar o Index e o Path ou quando há uma Remote Exception no **Search Module** para guardar o estado da **Dequeue**.

Todos os programas implementados contêm uma Thread especifica para apanhar os sinais Crtl-C. Esta Thread fica encarregue de encerrar corretamente cada um dos programas. Assim, quando este sinal é apanhado o método onCrash() é aplicado para guardar todos os dados gerados.

```java
try {
    Object readObject = ois.readObject();
    if (readObject instanceof URL) {
        URL url = (URL) readObject;
        storageBarrel.saveURL(url);
    } else {
        System.out.println("Barrel: The received object is not of type String!");
    }
} catch (ClassNotFoundException e) {
    System.out.println("Barrel: Error trying to read from Multicast Socket");
    storageBarrel.onCrash();
    return;
}
```

De forma a recuperar os pacotes perdidos no Multicast, sempre que um **Barrel** crasha e consequentemente há um timeout, o **Search Module** verifica qual **Barrel** ficou inativo para o remover da lista de **Barrels**, evitando futuros timeouts provocados por este crash.

Testes Realizados

Teste: Correr o programa com um Downloader e um Index Storage Barrel.

Estado: **Passou**

Descrição: O Sistema corre como suposto ainda que pouco eficiente, torna o Multicast desnecessário e todas as Queries são respondidas pelo mesmo Barrel.

Teste: Correr o programa com um Downloader e vários Index Storage Barrel.

Estado: **Passou**

Descrição: O Sistema passa a ter informação distribuída e os Barrels são escolhidos de forma circular para responder a Queries. O Multicast passa a ser necessário.

Teste: Correr o programa com mais que um Downloader e nenhum Index Storage Barrel.

Estado: **Passou**

Descrição: O Sistema funciona como suposto, porém não indexa nenhum URL novo. Para além disso, o Downloader fica em espera que haja pelo menos um **Barrel** ativo pois todos os dados que são gerados por ele nunca vão ser guardados.

Teste: Correr o programa com mais que um Cliente ativo.

Estado: **Passou**

Descrição: O Sistema devolve os resultados correspondentes ao Cliente que os pediu.

Teste: Verificar se o Protocolo Multicast realmente faz o retry.

Estado: **Passou**

Descrição: Forçou-se o encerramento um Barrel para este não enviar o *Acknowledgment*. Colocou-se um print no Timeout e verificou-se que este print ocorria as 3 vezes, pelo que o Downloader tentou reenviar as outras 2 vezes.

Teste: Verificar se o quando há um timeout, o Search Module faz um ping para todos os **Barrels**.

Estado: **Passou**

Descrição: Forçou-se o encerramento um **Barrel** para forçar um timout. O **Downloader** informa o **Search Module** que por sua vez pinga todos os **Barrels** da sua lista. Prints e debugs confirmam que o **Barrel** já não estava ativo, retirando-o da lista.

Teste: Testar o login de um utilizador com username e password certas.

Estado: **Passou**

Descrição: O Cliente consegue logar-se corretamente.

Teste: Testar o login de um utilizador com username ou password erradas.

Estado: **Passou**

Descrição: O Cliente depara-se com uma mensagem de erro diferente se o username não existir ou se a palavra pass estiver errada.

Teste: Testar o Cliente indexar um URL e pesquisar uma palavra contida nele.

Estado: **Passou**

Descrição: É possível pesquisar por uma palavra contida no URL indexado anteriormente e ver esse mesmo URL na página dos resultados da pesquisa.

Teste: Testar o Cliente indexar um URL e pesquisar uma palavra não contida nele.

Estado: **Passou**

Descrição: Ao pesquisar uma palavra que não esteja contida no URL indexado anteriormente, este URL não aparece nos resultados da pesquisa.

Conclusão

Ao longo do projeto surgiram dificuldades que após o diálogo entre os membros do grupo foram atenuadas e/ou resolvidas. Por vezes a solução para certos problemas não foi a mais obvia mas com esforço e dedicação chegou-se à solução pretendida. Com isto em vista, a distribuição das tarefas por elemento do grupo foi efetivamente organizada. Ambos trabalhamos um pouco em todas as funcionalidades pois achámos que seria vantajoso os dois elementos do grupo conhecerem ao pormenor todos os programas criados bem como todas as funcionalidades implementadas. Na última semana esta divisão alterou-se ligeiramente: enquanto o Eduardo aperfeiçoou o protocolo Multicast implementado, o Fábio deu mais atenção à escrita do relatório.
Em suma estamos contentes com o trabalho desenvolvido e com entusiasmados para a próxima meta, esperando que corra tão bem ou melhor que a primeira.