Meta 1 – Googol

Sistemas Distribuídos

FCTUC – Departamento de Eng. Informática 2023/24

Trabalho realizado por:

- Johnny Fernandes 2021190668
- Miguel Leopoldo 2021225940

Índice

Introdução

No atual cenário da internet, a vasta quantidade de informações disponíveis torna indispensável a existência de mecanismos de pesquisa eficazes e abrangentes a todo espaço web. Este projeto surge no âmbito da disciplina de Sistemas Distribuídos e tem o propósito de desenvolver um motor de busca de páginas web de nome "Googol", abrangendo funcionalidades essenciais para indexação e pesquisa, inspirado em serviços reconhecidos como o *Google*, *Bing*, *DuckDuckGo*, entre outros.

O objetivo primordial deste projeto será, portanto, criar um sistema em *Java* capaz de realizar a indexação automática de páginas web, utilizando um *Crawler* com recurso à biblioteca *JSoup*, e disponibilizar uma interface de pesquisa que retorne resultados relevantes para as consultas dos utilizadores (nesta primeira fase apenas faremos o sistema de indexação, deixando a interface para a segunda fase de entrega). Cada página a ser indexada conterá informações cruciais, como URL, título, descrição e citações de texto (vulgo *tokens*), bem como outras ligações presentes na própria página. Estes dados serão utilizados em diversas fases de processamento por forma a melhorar a experiência do utilizador.

Ao efetuar uma pesquisa, os utilizadores terão acesso a uma lista de páginas que contenham as palavras-chave pesquisadas, ordenadas por relevância, de acordo com critérios estabelecidos pelo sistema. Além disso, os utilizadores terão a possibilidade de sugerir *URLs* para inclusão no índice do sistema, contribuindo assim para a expansão e atualização contínua da base de dados.

Um aspeto-chave deste projeto é a capacidade de realizar indexação recursiva ou iterativa, explorando todas as ligações encontradas em cada página inicialmente indexada. Isso permitirá uma cobertura abrangente da web, garantindo que o motor de busca seja capaz de descobrir e indexar novos conteúdos de forma contínua.

De seguida e ao longo do relatório iremos explorar a arquitetura do software a ser desenvolvido, explicando com detalhe as decisões que nos levaram a essa escolha. Detalharemos o funcionamento da interação entre os diferentes elementos do sistema, através dos componentes *Multicast* e *RMI*. Além disso iremos mostrar os resultados obtidos dos nossos testes de funcionalidade.

Por fim e não menos importante, faremos uma exploração breve sobre a divisão do trabalho e a justificação para essa separação e divisão de tarefas pelos dois elementos que compõe o projeto.

Assim, e em resumo, este projeto visa desenvolver um motor de busca web robusto e eficiente, que atenda às necessidades dos utilizadores ao proporcionar acesso rápido e relevante às informações disponíveis na internet.

Arquitetura do Software

Nesta secção do relatório iremos definir as nossas escolhas em relação à arquitetura do software desenvolvido. Importa definir que estas escolhas surgem, em parte, do seguimento do conhecimento adquirido ao longo do semestre na disciplina, e visam o bom funcionamento da aplicação num cenário que, não sendo de produção, se pretende que seja o mais próximo possível de um caso real. Alguns dos elementos abaixo abordados não são de avaliação ou uso obrigatório, no entanto, cremos que a sua aplicação ajude a conseguir performance adicional ao software bem como a obtenção de melhores resultados de pesquisa.

Deste modo, iremos dividir esta secção nos vários componentes isolados que funcionarão como aplicações individuais na rede e que, no seu conjunto, formarão o bom funcionamento do motor de busca, nomeadamente:

- Gateway – Permite o controlo do cliente com o servidor
- *URL Queue* – Permite guardar os *URLs* a serem futuramente indexados
- *Downloaders* – Fazem o *crawling* das páginas web existentes na *URL Queue*
- Index *Storage Barrels* – Funcionam como a base de dados, fazendo também o processamento dos dados
- Cliente c/ painel de administrador – Permite a interface *CLI* para a utilização do motor de busca, além de permitir obter várias métricas sobre o sistema, atualizadas em tempo real

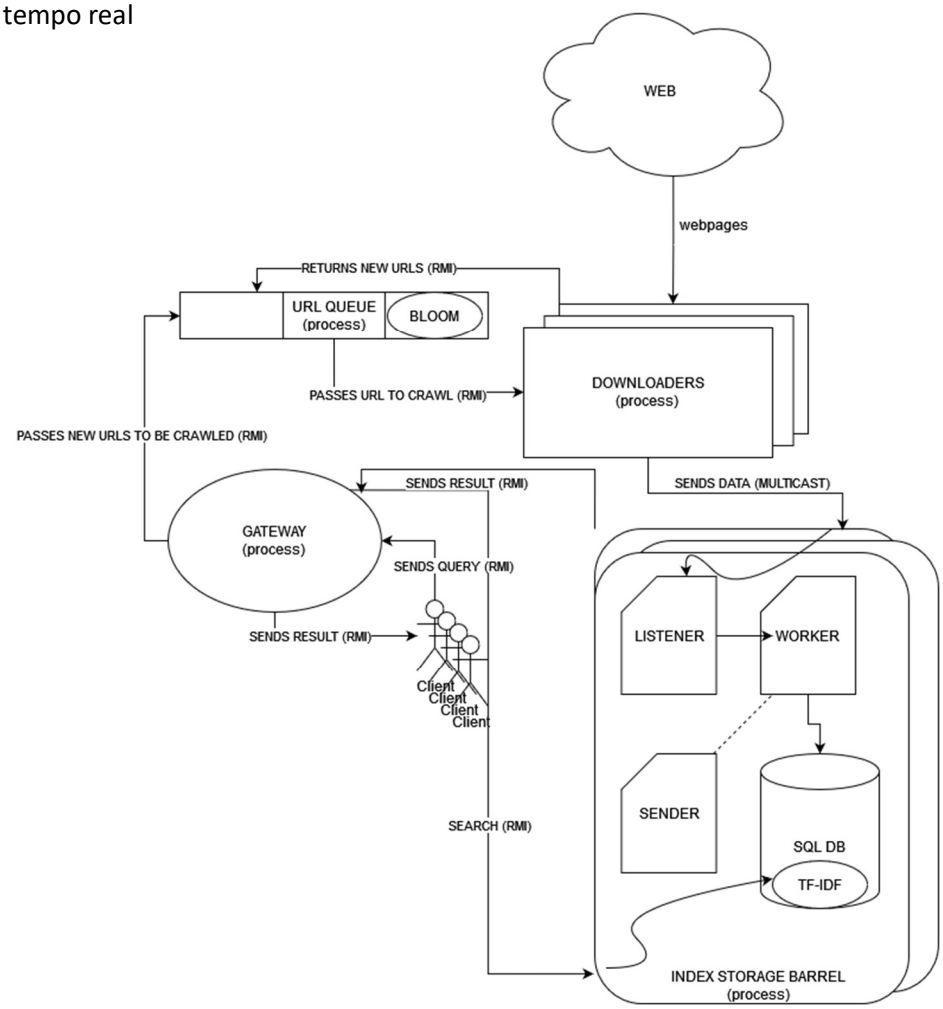

Figura 1 - Overview da arquitetura

Assim, passemos à explicação de cada componente.

Gateway

A *Gateway* é o principal processo do sistema visado neste projeto. Este interage como um controlador da informação entre os *ISB* (onde se incluem as bases de dados) e os clientes. Quando um cliente requisita informação seja ela em forma de palavras (*tokens*) ou *URLs*, a *Gateway* define se o pedido deverá ser usado para transmitir a *URL* diretamente à *URL Queue* [por *UDP Multicast*] ou se deverá proceder a uma pesquisa na base de dados nos *ISB* [por *RMI*] (a seleção do *ISB* a ser usado é feita de forma aleatória entre o grupo de *ISB* disponíveis).

Por outra via, o *Gateway* é responsável por atualizar a informação em tempo real, que será mostrada diretamente ao cliente em caso de acesso à área de administração.

Convém denotar que, em caso de falha, não haverá qualquer tipo de funcionamento, tendo sido tratada a questão das potenciais falhas. Numa situação deste tipo o terminal cliente informará o utilizador sobre insucesso na obtenção de acesso ao sistema. Como o terminal de cliente está sempre dependente deste, não teremos um sistema funcional. Contudo, pressupondo a persistência na restante ligação dos componentes, a indexação prosseguirá o seu curso normal. A comunicação da *Gateway* com os restantes elementos é feita por *RMI*.

URL Queue

A *URL Queue* é uma estrutura do nosso projeto, que funciona como um serviço isolado dos outros (processo independente) e que comunica com os *Downloaders* [por *RMI*] e com a *Gateway* [por *UDP Multicast*], e está projetada para armazenar os *URLs* a serem indexados e disponibilizá-los aos *Downloaders*.

Esta fila de *URLs* (`URLQueue`) desempenha um papel fundamental no processo de indexação, pois é a partir dela que os *Downloaders* (`Downloader`) obtêm os *URLs* a ser indexados. Quando um *Downloader* está pronto para processar um novo URL, solicita à *URL Queue* o próximo URL disponível. A implementação da *URL Queue* é feita com recurso a uma `BlockingQueue` e com utilização de um *Bloom Filter (*`BloomFilter`*)* para uma maior eficiência, conforme veremos de seguida.

Em caso de falha, os *Downloaders* detetarão a falha da *URL Queue*. No caso da inserção de um URL pelo cliente, este será avisado do insucesso da tentativa.

Cada *Downloader*, com seu conjunto de *Workers* individuais (*threads*), pode solicitar *URLs* à *URL Queue* de acordo com o seu estado atual, sendo que estes serão disponibilizados se assim for possível e a *URL Queue* tiver ainda elementos para indexação, funcionando numa lógica FIFO tanto para a saída dos *URLs* como para a ordem dos pedidos. Isto permite uma adaptação dinâmica à carga de trabalho e garante um processamento equilibrado e eficiente dos *URLs*.

Bloom filter

Conforme visto acima, foi feita a adição de um algoritmo *Bloom Filter* (de classe `BloomFilter`) ao nosso projeto com o propósito de otimizar o processo de adição de *URLs*, em relação à verificação da existência de *URLs* já indexados. Esta estrutura de dados probabilística permite uma verificação rápida e eficiente da presença de um elemento num conjunto de dados, com um baixo consumo de memória e com a definição da probabilidade de erros associada (no nosso caso com a atribuição de 1% de taxa de erro).

Assim, no contexto do nosso projeto, o *Bloom Filter* é unicamente utilizado para verificar se um URL já foi indexado antes de ser adicionado, para ajudar a evitar a indexação duplicada de páginas da web e a melhorar a eficiência do sistema como um todo.

A implementação do *Bloom Filter* foi feita no decorrer do projeto, tendo-se optado pelo seu desenvolvimento sem recurso a bibliotecas como o *Guava*, e feito segundos os traços gerais que conseguimos apurar após pesquisa. Usa para isso um conjunto de *seeds* utilizadas numa função de *hashing*. Cremos que a implementação deste algoritmo ajudou de forma significativa à performance do funcionamento da nossa *URL Queue*.

Downloaders

Os *Downloaders* constituem outra das componentes do nosso projeto, e neste caso são responsáveis por fazer o download e análise das páginas web, através da extração de informações relevantes, como palavras (*tokens*) e *URLs*.

Estes *Downloaders* operam como um processo que cria várias *threads* `DownloaderWorker`, cada uma encarregada de processar uma página web individualmente a cada instante. Cada *worker* tem a tarefa de extrair todas as informações pertinentes de uma página web e organizá-las em objetos `CrawlData`. Estes objetos contêm campos como *URL*, título, descrição, *tokens* e *URLs* adicionais encontradas na página. Uma vez reunidas as informações nos objetos `CrawlData`, estes são comprimidos utilizando o *GZIP* e divididos em fatias de 1024 bytes. Estas fatias são então encapsuladas num objeto do tipo `Container`, que tem uma funcionalidade análoga à de um pacote de rede. Este encapsulamento é necessário para preparar os dados para a transmissão, onde o restante do protocolo *multicast* entra em ação para garantir a fiabilidade da mesma, conforme veremos abaixo em maior detalhe no capítulo de *Reliable Multicast*.

Este protocolo desenvolvido trata da transmissão dos dados, incluindo o armazenamento da informação enviada para possibilitar a retransmissão em caso de perda. Isto assegura que os dados sejam transmitidos de forma fiável e eficiente, mesmo em ambientes propensos a falhas ou perdas de pacotes.

Além disso, os *Downloaders* trabalham em conjunto com a *URL Queue*, como mencionado anteriormente, para obter as *URLs* a serem processadas. À medida que um *worker* fica disponível, é feita uma nova requisição de endereço à *URL Queue* para obtenção de novo *URL* a indexar.

Index Storage Barrels

Os Index Storage Barrels têm um dos papéis mais importantes do nosso sistema de motor de busca e são responsáveis pelo armazenamento e organização meticulosa das informações indexadas das páginas web, num sistema de base de dados conforme iremos explicar de seguida. Estes barris de armazenamento são projetados para conter uma variedade de informações sobre as páginas web indexadas, conforme já vimos anteriormente, além de outras meta-informações relevantes, como o número de atualização de cada operação (usado para o processo de sincronização de barris) e o valor de TF-IDF (a ser explicado abaixo).

Estes *ISB* são replicados por mais que uma instância e entram em processo de sincronização no momento de início, usando o protocolo *Reliable Multicast* criado, e explicado com mais detalhe no capítulo seguinte. Esta sincronização é verificada em comparação com a informação da última adição de novos dados (`id` das inserções na tabela `websites`) conjuntamente com as atualizações feitas aos dados (com base na coluna `updates` da tabela websites). Assim que alguma diferença for detetada, os dados serão transmitidos aos outros *ISB* desatualizados. Quando um *ISB* recebe esta informação, atualiza os seus dados com base nos que recebeu do outro *ISB*, dando assim prioridade aos dados que forem recebidos da sincronização, em relação aos dados a serem recebidos dos *Downloaders*.

É também neste processo, à semelhança do que acontece com o processo dos *Downloaders* que é utilizado o protocolo de *Reliable Multicast*, fazendo uma escuta constante ao grupo de *multicast* para recebimento de novas informações sobre os sites indexados. O *ISB* procede então ao processamento das informações, calculando determinados parâmetros, nomeadamente o valor de TF-IDF para cada *token* associada a cada documento e guarda os dados na base de dados. Por fim, ficam também os *ISB* sempre disponíveis para qualquer pedido feito por parte do Gateway para a recuperação de dados, através de *RMI*. Estes dados podem ser pesquisados tanto usando a ordem de maior quantidade de *URLs* como de TF-IDF mais elevado.

Os *ISB* representam um papel fundamental pois garantem a integridade dos dados de todo o sistema e a sua sincronização e consistência de estados é fundamental.

SQLite

Neste projeto optamos pela escolha do *SQLite* para os *Index Storage Barrels*, e a decisão baseou-se na simplicidade, eficiência e suporte à linguagem *SQL*. Além disso, o *SQLite* não requer um servidor separado, o que simplifica a implementação através do uso da biblioteca para Java. Por fim esta decisão acarreta ainda questões como a confiabilidade, portabilidade e acesso aos dados. Quando comparados com modelos mais simples como ficheiros de texto ou ficheiros de dados, ou ainda como bases de dados mais complexas como o *PostgreSQL*, o *SQLite* apresenta-se como uma solução mais adequada.

Além do mais o *SQLite* permite uma pesquisa num volume alargado de dados de uma forma muito mais otimizada sem o utilizador final ter de considerar aspetos fundamentais como a performance ou o acesso aos mesmos dados em simultâneo, não havendo a necessidade de aplicação de *locks* ou outros mecanismos que salvaguardem a funcionalidade e operacionalidade da base de dados.

TF-IDF

O TF-IDF é calculado considerando dois fatores principais:

- Frequência do Termo (*TF - Term Frequency*): Este fator mede a frequência com que uma palavra específica ocorre num documento. Quanto mais vezes uma palavra ocorre em um documento, maior é a sua relevância para esse documento.
- Frequência Inversa do Documento (*IDF - Inverse Document Frequency*): Este fator avalia a raridade de uma palavra em toda a coleção dos documentos. As palavras que aparecem em muitos documentos são consideradas menos importantes, enquanto aquelas que aparecem em poucos documentos são consideradas mais relevantes.

No nosso projeto, o TF-IDF é aplicado durante o processo de indexação das páginas web. Para cada palavra-chave extraída de uma página, calculamos o TF-IDF para determinar a sua relevância em relação a essa página específica e à coleção de documentos como um todo. As palavras com TF-IDF mais altos são consideradas mais importantes e estes são armazenadas nos Index *Storage Barrels* juntamente com outras informações relevantes da página para que, aquando da extração de resultados para o utilizador final, se possa apresentar os resultados com os resultados mais adequados à pesquisa feita.

Cliente e painel de administrador

A aplicação do cliente é a única interface (*CLI*) presente nesta primeira fase do projeto, havendo planeada uma futura integração de interface web. Deste modo, implementou-se um cliente elementar que é capaz de, através de *RMI*, contactar diretamente com a Gateway para a submissão de pedidos e posterior obtenção de dados, constantes nos *ISB*. Quando um cliente submete um endereço *URL* diretamente, este é diretamente reencaminhado [por UDP Multicast] para a *URL Queue*, a fim de ser indexado, processado e armazenado. Por outra via, quando são submetidas palavras, o resultado é pesquisado diretamente na base de dados, fazendo uma apresentação por ordem de página com mais ligações (de forma decrescente). É ainda possível outro modo de apresentação, através de um somatório dos valores de TF-IDF de cada palavra e apresentando-os ao utilizador final (cliente) de forma ordenada pelo nível de adequação.

Além disso, de considerar que o cliente possui também acesso a uma área onde é possível obter métricas do sistema. Este painel é acessível a todo o tipo de clientes, não sendo necessária uma autenticação e será capaz de mostrar as seguintes informações: o TOP 10 de pesquisas mais comuns, quais os *ISB* ativos naquele instante e o tempo médio de resposta de cada *ISB*, medido a partir da *Gateway*.

Este será o único meio de acesso ao sistema e acessível ao utilizador final e, por isso, em caso de inconsistências na execução do sistema, será alertado para a quebra de serviço.

Comunicação

A comunicação é uma das partes fundamentais do desenvolvimento deste projeto, havendo um forte foco na implementação de sincronização de componentes bem como dos métodos de comunicação utilizados (*RMI* e *Reliable Multicast*). Nesse sentido, o presente capítulo pretende dar uma vista aprofundada sobre as soluções encontradas para os problemas bem como a forma como se conseguiu garantir a integridade dos dados aquando do processo de indexação e sincronização.

Reliable Multicast

Nesta secção, uma das mais densas e mais fortemente desenvolvidas nesta meta do projeto, iremos explicar como foi concebido e desenvolvido o nosso protocolo de *multicast* confiável (*Reliable Multicast*), onde foram consideradas diversas etapas e desafios para garantir a transmissão eficiente dos dados.

Primeiramente, foi necessário garantir que a transmissão feita de *Downloaders* para os *Index Storage Barrels* (onde a implementação era obrigatória conforme o enunciado) fosse de tal forma fidedigna que apenas se considerariam dois cenários possíveis: todos os *ISB* a receber a mesma informação, ou então nenhum a receber a informação (caso em que nenhum está ativo).

Além do mais, para a existência de um menor *overhead*, foi descartada a necessidade de ACK (*acknowledgement*/confirmações) em detrimento da implementação NACK (negative *acknowledgement*/aviso de falha). Isto permite um menor uso da rede no envio de informação de recebimento dos pacotes, e ajuda a eliminar a redundância de informação uma vez que é possível fazer este processo de forma implícita.

Assim, com base nestes dois critérios iniciais essenciais, desenvolvemos um sistema de tal forma generalista que foi possível alagar a sua utilidade a outros cenários ao longo do projeto. Um dos planos para o protocolo foi não só ser capaz de receber qualquer tipo de objeto ou estrutura de dados, passado como *slices array* de até 1024 *bytes* por pacote, mas também que o uso do sistema fosse de uma certa forma transparente ao utilizador final (em semelhança ao *RMI*).

Do ponto de vista do código, o protocolo desenvolvido possui uma classe `ReliableMulticast`, sendo o objeto principal e que foi projetado para encapsular um conjunto de funcionalidades essenciais, nomeadamente um `Sender`, um `ReceiverListener` e um `ReceiverWorker`, cada um a operar como thread independente.

Além da classe `ReliableMulticast` de funcionamento do nosso protocolo, duas outras classes principais do tipo `Objects` desempenham um papel fundamental neste processo de transmissão de dados entre os *Downloaders* e os *ISB*: `CrawlData` e `Container`. A classe `CrawlData` armazena as informações relevantes sobre a indexação de um website, como *URL*, título, descrição, lista de *URLs* e lista de palavras (*tokens*), que é posteriormente comprimida utilizando o algoritmo GZIP e enviada em fatias de 1024 bytes dentro do campo "data" de um `Container`. Este último contém meta dados importantes, como `dataType`, `dataID` (*hash* do

objeto), número do pacote, número total de pacotes e uma indicação se é um pedido de retransmissão ou não.

Nas componentes do sistema `ReliableMulticast`, o `Sender` é o elemento responsável pela leitura de duas estruturas: o `senderBuffer` e o `retransmitBuffer` - este último utilizado para armazenar os dados enviados (e necessários em caso de retransmissão) e funciona como uma fila circular para que haja a rotação deste buffer de acordo a progressão gradual da indexação. O `Sender`, em contínua verificação destes buffers, envia para o destinatário do objeto da classe `Container` (explicado abaixo) o pedido. Todos os pacotes normais para envio são colocados no `senderBuffer` e os que foram enviados serão colocados no `retransmissionBuffer`. A justificação para esta separação prende-se com o facto de que uma está estritamente ligado com o local onde é implementado (`senderBuffer`) enquanto o outro (`retransmissionBuffer`) está intrinsecamente ligado ao protocolo em si e à sua confiabilidade, nunca podendo ser o mesmo buffer.

Por outro lado, a classe `ReceiverListener` faz escuta continuamente ao grupo *multicast*, colocando no `listenerQueue` (buffer) todos os pacotes recebidos, que serão posteriormente processados pelo `ReceiverWorker`. Esta última classe é a responsável pela maioria do funcionamento do nosso protocolo, garantindo que todos os pacotes sejam recebidos e processados corretamente. O `ReceiverWorker` utiliza um `HashMap` para rastrear a receção de cada pacote e verificar se todos os pacotes foram recebidos. Em caso de perda de pacotes, é solicitada a retransmissão ao `Sender`. Além disso, o `ReceiverWorker` reconstrói o objeto inicialmente enviado após a receção de todos os pacotes, garantindo a integridade dos dados. Os últimos pacotes perdidos são reverificados e é pedida uma retransmissão a cada 5 segundos em caso de perda.

Assim, e em resumo, o protocolo *Reliable Multicast* foi cuidadosamente desenvolvido para garantir a transmissão eficiente e fiável de dados em ambientes *multicast*. Desde a sua conceção até à implementação, cada aspeto foi pensado levando em consideração as necessidades específicas do projeto, resultando num sistema que é versátil e robusto, e que pode ser utilizado em vários cenários ao longo do projeto, tal como foi usado na sincronização de *ISB*.

RMI

O *RMI* é um mecanismo fornecido pelo Java, para a comunicação entre processos em diferentes máquinas (*JVMs*), e que permite que os métodos dos objetos remotos sejam invocados. Isto é feito de uma forma transparente para o utilizador final, e acabou por servir de modelo para a implementação do protocolo *Reliable Multicast* conforme visto anteriormente, no que toca à simplicidade de utilização. Fazemos este acesso de métodos remotos através das interfaces `RMIGatewayInterface`, `IndexStorageBarrelInterface` e `URLQueueInterface`.

No nosso projeto, o *RMI* é utilizado na maioria das comunicações, nomeadamente entre o cliente e o Gateway, o Gateway e os *ISB* e por fim entre os *Downloaders* e a *URL Queue*.

Testes de software

Os testes permitem identificar e corrigir erros e falhas numa fase inicial do processo de desenvolvimento, reduzindo assim o tempo necessário para resolver problemas mais tarde. Além disso, os testes garantem que o software atenda aos requisitos funcionais e não funcionais estabelecidos. Por esse motivo e pela garantia de funcionalidade necessária para que o desenvolvimento de toda a aplicação se torne possível, implementou-se três conjuntos de testes, descritos abaixo e que fazem acompanhar o código-fonte do projeto.

Protocolo Multicast

Nesta fase foi criado um ficheiro de teste de funcionalidade (`ProtocolTester`) para verificação se os diferentes componentes do protocolo *Reliable Multicast* estão funcionais. No caso de teste de falhas e recuperação de dados, foi feita uma implementação temporária diretamente no protocolo para que fossem perdidos pacotes de forma proposital (com recurso ao *Random*). Através desse teste foi possível apanhar um conjunto de inconsistências e gradualmente adaptou-se o código para a obtenção de um resultado completamente funcional e que permite a passagem de informação de uma máquina para outra, independentemente do objeto ou conjunto de dados que se pretenda transmitir.

Abaixo deixamos a tabela com um conjunto de fases de teste e os seus respetivos resultados.

Tabela 1 - Resultados do dos testes ao Reliable Multicast

FUNCIONALIDADE TESTADA	PASS
Inicialização e encerramento do objeto	PASS
Inicialização das *threads Sender*, *ReceiverListener* e *ReceiverWorker*	PASS
Funcionamento do protocolo para envio e recebimento	PASS
Perdas de pacotes [taxa de erro a 5% diretamente na implementação]	PASS
Recuperação dos pacotes perdidos com recurso à componente *reliable*	PASS
Tamanho do buffer de leitura ajustado à realidade do objeto Container	PASS

Index Storage Barrels

No que toca aos *Index Storage Barrels*, foi desenvolvido também um ficheiro de teste de funcionalidade que se faz acompanhar junto do código-fonte (`StorageTester`) para verificação da funcionalidade dos mesmos, nomeadamente recebimento, tratamento, inserção e devolução de dados.

Convém indicar que todo este tratamento e teste foi essencial para garantir o bom funcionamento do sistema e para melhoria e adaptação das *queries* utilizadas para conseguirmos ganhos de performance e velocidade.

Tabela 2 – Resultados dos testes aos Index Storage Barrels

FUNCIONALIDADE TESTADA	PASS
Recebimento dos pacotes através dos Downloaders por Reliable Multicast	PASS
Unpackaging do conteúdo dos pacotes enviados, sem perdas de informação	PASS
Tratamento da informação (TF-IDF, contagem de tokens, entre outras operações)	PASS
Armazenamento da informação na base de dados em SQLite	PASS
Sincronização em espelho com os *ISB* adjacentes quando inicializados	PASS
Resultados apresentados ao utilizador final	PASS

Convém considerar que o TF-IDF tendo sido implementado diretamente nos *ISB* foi também, por isso, sujeito a testes aquando do cálculo e da inserção dos seus valores, na base de dados, não tendo, por isso, sido feito um teste específico para o uso desta ferramenta.

Bloom Filter

Por fim, outro dos testes implementados foi o *Bloom Filter* que, considerando o seu tipo de utilização e a forma como foi implementado no mecanismo da *URL Queue*, foi da máxima importância o seu teste para garantir a funcionalidade, nomeadamente a não inserção de dados repetidos na *queue*, bem como a não perda da informação dos *URLs* em potenciais casos de falsos negativos (como os casos em que o algoritmo poderia indicar que a *URL* já estaria presente na *queue* sem estar verdadeiramente, perdendo-se assim o endereço).

Deste modo deixamos abaixo uma breve tabela sobre os casos avaliados e sobre o seu resultado.

Tabela 3 – Resultados dos testes ao Bloom Filter

FUNCIONALIDADE TESTADA	PASS
Cálculo correto das *hashs*	PASS
Inserção de dados na *queue* com base no *Bloom Filter*	PASS
Verificação com vários *URLs* aleatórios	PASS
Repetição de *URLs* para garantir a sua presença na *queue*	PASS
Inserção de novos *URLs* para garantir a sua não presença na *queue*	PASS

Distribuição de tarefas

Por fim, de igual forma relevante, procedemos à demonstração da divisão de tarefas ao longo do desenvolvimento do projeto. Este projeto foi desenvolvido por dois alunos, pelo que, a boa divisão de tarefas foi fundamental para atingir os objetivos propostos. Segue abaixo uma tabela demonstrativa da divisão das tarefas, tendo havido uma proximidade em termos do tempo despendido com uma forte comunicação e presença ao longo de todo o desenvolvimento.

Johnny Fernandes	**Miguel Leopoldo**
Desenvolvimento geral do *Reliable Multicast, nomeadamente conceção da arquitetura, objeto Container, transmissão de dados. Criação do ficheiro de teste de funcionalidade*	Correção de lógica, defeitos, e melhor implementação dos casos *reliable* para garantia de não perda de pacotes
Desenvolvimento e arquitetura dos *Index Storage Barrels* e definição das *queries* de acesso à base de dados, nomeadamente inserção e recuperação de dados, e cálculo do TF-IDF. Criação do ficheiro de teste de funcionalidade	Desenvolvimento dos *Index Storage Barrels* com particular foco na conectividade, implementação e garantia da sincronização e tratamento de erros, bem como conexão ao Gateway
Desenvolvimento do Cliente e *Admin Console, em particular a forma como é feita a leitura de inputs do utilizador, e implementação do Admin Console*	Desenvolvimento geral do Cliente e melhorias/alterações feitas à forma como o *Admin* Console acede aos dados dos *Barrels*
Desenvolvimento do *Bloom Filter* e dos seus testes de funcionalidade	Desenvolvimento do Gateway para comunicação entre o *Cliente*, os *ISB* e a *URL Queue* (onde se inclui a aplicação de *UDP Multicast*)
Testes de conectividade de *multicast*, experimentação e *debug* de redes e interfaces de rede remotamente (com recurso ao *ZeroTier*)	Tratamento de falhas e controlo de situações que possam gerar crashs acidentais em casos de perda de conectividade
Geração e verificação/limpeza do *Javadoc*	Criação do *Javadoc*
Produção do relatório final	Melhorias finais e polimento

A informar que além desta divisão de tarefas, houve um forte empenho de ambos, com comunicação quase diária e *peer programming* numa boa porção do tempo aplicado ao projeto, conseguindo por isso chegar a várias soluções práticas e obtendo-se bons resultados no funcionamento da aplicação. Houve vários desafios enfrentados, em particular no que toca ao uso das redes *multicast* e *RMI*, sendo necessário gastar uma quantidade de horas imprevistas em processo de *debug* e análise da rede com recurso a várias estratégias, nomeadamente alteração das interfaces de rede, uso do *Wireshark*, entre outras.

Assume-se que o tempo aplicado ao projeto, de forma síncrona e assíncrona foi bastante semelhante em ambos os casos.

Conclusão

Em conclusão, este projeto permitiu-nos aprofundar os nossos conhecimentos do funcionamento de redes de comunicação através do uso de protocolos e métodos conhecidos e bastante utilizados, como é o *Multicast* (onde se criou uma versão *reliable* do protocolo) e o RMI (onde se faz invocação de métodos remotos), descobrindo assim como utilizar a programação de sistemas distribuídos para a criação de um ecossistema de aplicativos que, juntos, dão a um utilizador final ferramentas bastante úteis, como é o caso de um motor de busca, objeto de desenvolvimento neste projeto.

Apesar da aplicação da interface de programação RMI em Java, percebe-se a utilidade geral e o modo como, de igual forma para outros modelos RMI de outras linguagens de programação, se pode aceder a métodos remotos para execução de tarefas remotamente e obtenção de resultados, como se de uma máquina local se tratasse. O RMI traz-nos imensas vantagens como a sua transparência e facilidade de uso, integração direta com o Java (neste caso), a segurança e o seu desempenho.

Por outra via o *Multicast*, na sua implementação *reliable* permitiu também encontrar formas muito leves de transmitir dados e de forma fiável, havendo retransmissão de pacotes perdidos assim que necessário e havendo um *overhead* menor face a outros tipos de comunicação. Acreditamos que a implementação do *Reliable Multicast* no decorrer deste projeto tenha chegando a bom porto no seu desenvolvimento, uma vez que, além de funcional é também bastante versátil permitindo a transmissão de quaisquer tipos de dados, sendo necessários poucos ajustes para melhoria de performance e fiabilidade.

Por fim, a implementação de mecanismos como o *Bloom Filter* e o TF-IDF foram também importantes para melhor compreender os mecanismos que estão por trás de um motor de busca e de outras ferramentas de recuperação de dados textuais, como é o caso do TF-IDF ou de melhores formas de inserção de dados em listas, filas ou pilhas, como é o caso do *Bloom Filter*.

Em nota de rodapé, destacamos que apesar do projeto nos ter proporcionado esta compreensão nestas diferentes tecnologias, o mesmo ainda poderá sofrer alterações e melhorias até à apresentação da segunda fase do projeto, levando a melhores performances e resultados.