

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Arthur Rodrigues Cruz

**Implementação de um Framework para
Desenvolvimento de Sistemas de Recuperação
da Informação Distribuídos**

Uberlândia, Brasil

2018

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Arthur Rodrigues Cruz

**Implementação de um Framework para Desenvolvimento
de Sistemas de Recuperação da Informação Distribuídos**

Trabalho de conclusão de curso apresentado
à Faculdade de Computação da Universidade
Federal de Uberlândia, Minas Gerais, como
requisito exigido parcial à obtenção do grau
de Bacharel em Sistemas de Informação.

Orientador: Prof. Dr. Rodrigo Sanches Miani

Universidade Federal de Uberlândia – UFU

Faculdade de Ciência da Computação

Bacharelado em Sistemas de Informação

Uberlândia, Brasil

2018

Resumo

Este trabalho consiste na arquitetura e implementação de um *framework* que permita ao usuário desenvolver um sistema ou subsistema de organização e recuperação da informação distribuído. O objetivo é fornecer um nível de abstração suficiente para que seja possível modularização dos componentes. Desta forma, viabilizando implementações personalizadas de componentes, que podem ser facilmente acopladas e desacopladas ao sistema. Para isso, foi desenvolvido um conjunto de interfaces que regem a comunicação entre os sistemas e também foi criada uma camada de comunicação que gerencia a distribuição do sistema com uma rede *peer-to-peer* em anel. Além disso, é discutido uma implementação fornecida, pronta para uso, baseada no *framework* e um estudo de caso de cliente e servidor que exemplificam a utilização prática da ferramenta.

Palavras-chave: Sistemas de informação, Sistemas distribuídos, Recuperação da Informação.

Lista de abreviaturas e siglas

TREC	<i>Text REtrieval Conference</i>
RI	<i>Recuperação da Informação</i>
ORI	<i>Organização e Recuperação da Informação</i>
SD	<i>Sistemas Distribuídos</i>
URL	<i>Uniform Resource Locator</i>
SO	<i>Sistema Operacional</i>
MIT	<i>Massachusetts Institute of Technolog</i>
DORIF	<i>Distributed Organization and Retrieval of Information Framework</i>
TF	<i>Term Frenquency</i>
IDF	<i>Inverse Document Frequency</i>

Sumário

1	INTRODUÇÃO	6
2	REVISÃO BIBLIOGRÁFICA	8
2.1	Organização e Recuperação da Informação	8
2.1.1	Modelagem	9
2.1.2	Coleta	10
2.1.3	Indexação	10
2.1.4	Consulta	11
2.1.5	Função de Ranqueamento	12
2.1.6	Recuperação	13
2.2	Sistemas Distribuídos	13
2.3	Frameworks	14
2.4	Trabalhos Correlatos	15
2.4.1	Apache Lucene	15
2.4.2	Apache Solr	17
2.4.3	Vespa	17
2.4.4	<i>Carrot²</i>	17
2.4.5	<i>The Information Grid</i>	18
2.4.6	Comparação	19
3	DESENVOLVIMENTO	20
3.1	Arquitetura do Framework	20
3.1.1	Camada Lógica	21
3.1.1.1	Interfaces	21
3.1.1.2	<i>Collection Handler</i>	22
3.1.1.3	<i>Factory</i>	23
3.1.1.4	<i>Collector</i>	23
3.1.1.5	<i>Pre Processor</i>	24
3.1.1.6	<i>ITerm</i>	25
3.1.1.7	<i>IWeight</i>	25
3.1.1.8	<i>IDocument</i>	25
3.1.1.9	<i>IVocabulary</i>	26
3.1.1.10	<i>IQuery</i>	26
3.1.1.11	<i>IRankingFunction</i>	27
3.1.2	Camada de Comunicação	28
3.1.2.1	<i>Thrift</i>	28

3.1.2.2	<i>Chord</i>	29
3.1.2.3	<i>DORIF Handler</i>	32
4	RESULTADOS	35
4.1	Implementações Disponíveis	35
4.1.1	Decisões de Projeto	35
4.1.2	Componentes Nativos	35
4.1.3	Servidor <i>DORIF</i>	38
4.2	Estudos de Caso e Aplicação	38
4.2.1	Servidor: Coleção de Texto Simples	38
4.2.2	Cliente: Console	41
5	CONCLUSÃO	47
	REFERÊNCIAS	49

1 Introdução

A recuperação da informação tem um papel fundamental na computação desde muito antes dos atuais buscadores da Internet. Os primeiros sistemas de recuperação da informação construídos datam do final da década de 1940 ([SANDERSON; CROFT, 2012](#)). Entre a década de 1990 até o presente momento deste trabalho (2018), estes sistemas passaram a compor sistemas distribuídos, tendo como exemplo o [Google \(2018\)](#).

Tal sistema opera de forma distribuída em vários *Data Centers* espalhados pelo mundo, e atende a uma infinidade de usuários também de regiões distintas do planeta. Deste modo, é possível recuperar informações dos mais variados tipos, atendendo uma grande demanda de requisições de todo o mundo.

No entanto, implementar sistemas de Recuperação da Informação (RI) distribuídos não é uma tarefa trivial, devido às complexidades somadas de um sistema de RI e um distribuído. Neste caso, é necessário lidar com: abstrações de documentos, indexação, modelos de recuperação, rede, servidores, clientes, controle de concorrência, entre outros fatores.

Ainda assim, este tipo de software torna-se cada vez mais requisitado tanto em ambientes acadêmicos quanto em ambientes não formais, como no fornecimento de serviços de buscas para a sociedade. Devido a esta realidade, existe a necessidade da utilização de *frameworks*, como o [Apache Lucene \(2017\)](#), que simplificam a programação destes softwares.

Porém, ainda que estes *frameworks* facilitem o processo de implementação, os mesmos apresentam limitações intrínsecas a sua própria implementação. Um exemplo seria a disponibilidade de alguns modelos de ranqueamento pré-definidos, sem a possibilidade de uma implementação personalizada poder ser acoplada. Como o modelo de ranqueamento define a forma que o computador interpretará os documentos, as consultas e o cálculo da similaridade entre estes elementos, é possível que um modelo fechado não satisfaça determinadas peculiaridades de pesquisas ou aplicações comerciais. Isto poderia levar à necessidade de uma implementação completa de um modelo de RI.

O diferencial do *framework* consiste na flexibilidade de implementação de seus componentes e na facilidade de uso de suas interfaces, quando comparado a outras soluções do mercado. Há também, a simplicidade na criação e configuração de servidores, tornando a escalabilidade natural. A heterogeneidade nos tipos de arquivos de texto que podem ser indexados pelo *framework* é uma outra característica de flexibilização que difere do encontrado atualmente.

Este trabalho tem como objetivo desenvolver um *framework* para criação de sistemas de organização e recuperação da informação distribuídos, com um nível de abstração alto o suficiente para que o usuário final tenha a possibilidade de implementar módulos fracamente acoplados para todas as etapas de RI disponíveis na literatura. Além da flexibilidade fornecida, o *framework* será implementado sob uma arquitetura híbrida, podendo existir vários servidores que se comuniquem e processem requisições dos clientes de maneira distribuída.

O projeto foi implementado em Java 8 e utiliza o *framework* [Thrift \(2017\)](#). A utilização de ambos permite portabilidade entre sistemas e linguagens de programação, além da escalabilidade requerida por sistemas distribuídos. Foram desenvolvidos: uma biblioteca, que constitui o *framework* em si, um estudo de caso de servidor que implementará o *framework* e um estudo de caso de cliente que utilizará os serviços do servidor.

Este trabalho encontra-se organizado da seguinte maneira: o capítulo dois apresenta a revisão bibliográfica, englobando os principais conceitos e termos utilizados, além dos trabalhos correlatos; o terceiro capítulo expõe todo o processo de criação da arquitetura, implementações prontas fornecidas dentro do *framework* e os estudos de caso; o último capítulo trata da conclusão, remetendo o que foi efetivamente concluído, o que não coube no escopo do projeto, considerações finais e trabalhos futuros.

2 Revisão Bibliográfica

Este capítulo consiste em expor os principais conceitos utilizados e o estado da arte das áreas que compõem esta monografia: Organização e Recuperação da Informação (ORI), Sistemas Distribuídos (SD) e *Frameworks*. Ao final são apresentados alguns trabalhos correlatos e sua comparação com o proposto por este trabalho.

2.1 Organização e Recuperação da Informação

Depois do surgimento da escrita, dos meios necessários para se criar e manter registros e sua disseminação e adoção, se tornou complexo sanar a necessidade de se encontrar determinada informação nos textos. Adiante na história do homem, coleções crescentes de livros e outros documentos estavam à disposição, porém, saber onde, dentre as várias escrituras, se localizava determinado tema ou conhecimento específico era um trabalho árduo. Normalmente, havia um profissional responsável por gerenciar algum tipo de catálogo dos documentos.

Devido a essa complexidade nas buscas algumas máquinas começaram a ser desenvolvidas que permitiam, ainda que primitivamente, catalogar e encontrar livros de acordo com algumas consultas simples, como um tema ([EDWARD, 1920](#)). Após o surgimento dos computadores, logo, foi imaginado que os mesmos poderiam ser utilizados para otimizar a tarefa de armazenar, catalogar e recuperar registros como descrito por [Nanus \(1960\)](#).

Ao mesmo tempo, técnicas e métodos para se indexar, ou seja catalogar, documentos foram surgindo e sendo aperfeiçoadas. O procedimento clássico para classificação dos documentos era via classificação hierárquica de assuntos como proposto por [Dewey \(1899\)](#). Foi concebido por [Taube, Gull e Wachtel \(1952\)](#) que palavras poderiam ser associadas a um livro ou documento como forma de indexação e utilizadas para encontrá-los posteriormente.

Avanços na área de RI normalmente possuem como objetivo melhorar a forma como se aborda o problema fundamental da recuperação de informação. Este, consiste em maximizar o número de documentos relevantes, minimizar a quantidade de irrelevantes e ordenar a classificação por ordem decrescente de relevância, dada uma necessidade de informação de um usuário, como descrito em [Baeza-Yates e Ribeiro-Neto \(2013\)](#). Para alcançar tal objetivo, a RI moderna possui várias etapas. Todas contribuem para viabilizar ou melhorar alguma parte do processo. É possível fragmentar o método de RI de várias formas. Para este trabalho a seguinte divisão foi utilizada: Modelagem, Coleta, Indexação, Consulta, Função de Ranqueamento e Recuperação. Existem também as fases de expansão

e realimentação de consultas, porém, estas não são fundamentais para o procedimento e não serão abordadas neste trabalho. A Figura 1 apresenta um fluxograma do processo de RI, destacando as etapas citadas.

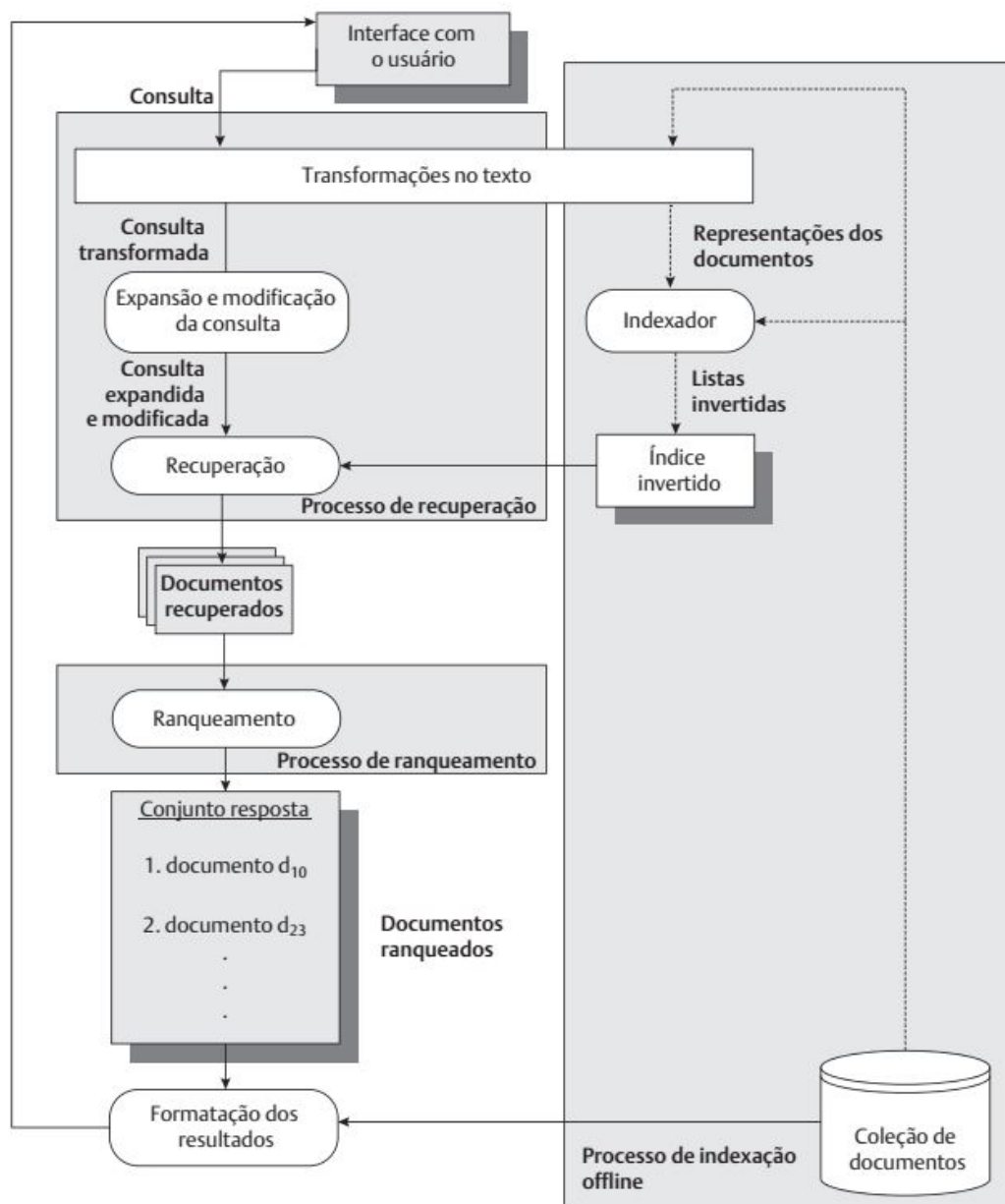


Figura 1 – Diagramas das etapas de indexação, recuperação e ranqueamento de documentos em RI. Retirado de (BAEZA-YATES; RIBEIRO-NETO, 2013).

Em sequência, serão abordadas e discutidas as etapas de: Modelagem, Coleta, Indexação, Consulta, Função de Ranqueamento e Recuperação.

2.1.1 Modelagem

A etapa de modelagem é uma parte anterior ao processo de organização e recuperação da informação em si. É neste ponto em que são definidos os modelos lógicos,

que são os responsáveis por representar os documentos e consultas. Também é definida a função de ranqueamento, utilizada para classificar os registros recuperados como mais ou menos relevantes. De acordo com [Baeza-Yates e Ribeiro-Neto \(2013\)](#) um modelo de recuperação de informação consiste em quatro itens, sendo eles: visões lógicas que representam os documentos de uma coleção, visões lógicas das necessidades de informação dos usuários (chamadas de consultas), uma visão, ou modelo, capaz de relacionar consultas e documentos e uma função de ranqueamento que gera um número real cuja interpretação é o grau de relacionamento entre documento e consulta. O ranking de documentos é decorrente de sua ordenação. No caso, o primeiro elemento sendo o documento mais relevante e o último o menos relevante.

Nos primórdios da computação haviam somente documentos textuais. Imagens, Áudios, Vídeos e páginas da Web surgiram conforme os computadores adquiriram capacidade, a passos rápidos, de lidar com estes formatos. Concomitantemente, a necessidade de busca nestes tipos de arquivos nasceu e cresceu juntamente com a popularidade dos mesmos. No entanto, recuperação de arquivos multimídia exige abordagens diferentes das utilizadas para texto não estruturado, como pode ser observado em [Enser \(2008\)](#). Logo, não serão utilizadas neste trabalho.

2.1.2 Coleta

Antes de recuperar qualquer informação, é necessário que exista uma base de documentos a ser consultada. Os documentos podem compor a base de diversas maneiras: podem estar no sistema de arquivos local, em um servidor remoto dentro da mesma rede, espalhados pela Internet como no caso das páginas web, entre outros. A fase de coleta consiste em analisar estes documentos onde quer que estejam e repassar todo ou parte de seu conteúdo à próxima fase, a Indexação.

2.1.3 Indexação

O processo de indexação varia de acordo com o modelo de RI utilizado, podendo ser necessário mais ou menos sub-processos para criar o mapeamento. Os modelos mais utilizados para indexação de texto não estruturado utilizam as palavras ou grupos de palavras, chamados de termos, de um documento como meio de indexá-lo.

É possível utilizar todo o documento ou parte do mesmo, como por exemplo os resumos (*abstracts*) de artigos, para extrair os termos. Durante a extração, é comumente aplicado algum tipo de normalização, ou pré processamento, com o intuito de reduzir a quantidade de termos muito comuns e que não tem valor descritivo, como por exemplo artigos (a, o, as, os). Remoção de caracteres especiais, acentuação, letras maiúsculas, entre outras técnicas, também são formas de normalização que auxiliam na obtenção de uma

melhor taxa de acertos nas consultas.

Na Figura 2 é representado um processo de indexação simples. Nele, o texto presente no documento é pré-processado, como citado anteriormente. Como resultado, obtêm-se termos que irão compor o índice. Este índice é um dicionário, onde a chave é o próprio termo e o valor é um objeto na memória que contém as informações de pesos e ponderações do respectivo termo.

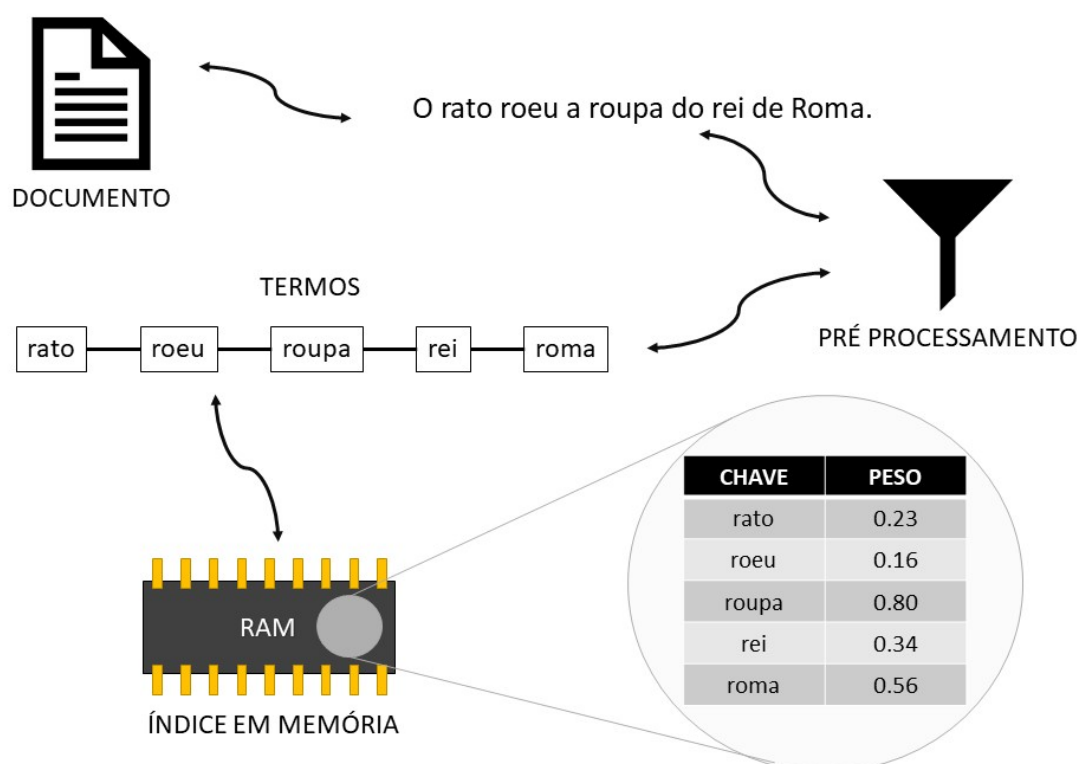


Figura 2 – Diagrama ilustrando o processo de indexação em RI.

2.1.4 Consulta

Uma vez que a coleção de documentos foi devidamente indexada, é possível submeter consultas. As consultas são a forma como os usuários do sistema de RI expressam suas necessidades de informação. O mesmo tratamento aplicado ao conteúdo dos documentos na etapa de indexação deverá ser executado na consulta, para garantir a coesão da comparação e também, aumentar as chances de sucesso na procura por similaridades.

Uma consulta é a representação de uma necessidade de informação que o usuário do sistema submete ao mesmo para que os documentos correlacionados sejam apresentados. A forma de estruturação de uma consulta pode variar de acordo com o modelo utilizado. No modelo booleano, por exemplo, utiliza-se palavras-chave que descrevem relacionamentos

lógicos entre palavras, como 'E', 'OU' e 'NÃO'. Um exemplo de consulta booleana seria "bola E água OU quintal NÃO rua".

Com a utilização de modelos de ranqueamento melhores a elaboração da consulta foi direcionada para um domínio mais próximo da fala. Os sistemas atuais, utilizando modelos como o vetorial, são capazes de processar perguntas naturais, com por exemplo "Qual a capital da Nicarágua?".

2.1.5 Função de Ranqueamento

Quando a consulta terminar de passar pelo mesmo pré-processamento realizado na indexação, a mesma será testada contra os índices dos documentos por meio da função de ranqueamento. Esta função, que varia de acordo com o modelo escolhido, é a responsável por gerar um número real que indicará o grau de similaridade entre a consulta e os documentos comparados. Uma boa função de ranqueamento é aquela que define um grau elevado de similaridade para um documento altamente relevante e penaliza os não relevantes. A escolha da função de ranqueamento é um dos pontos mais importantes do processo de RI.

Durante a evolução da RI, várias técnicas para funções de ranqueamento foram desenvolvidas. Usualmente uma técnica consiste na observação de alguma característica dos termos com relação ao documento. A quantidade de vezes que um termo aparece em um documento, *term frequency* (TF), é diretamente proporcional ao seu peso. Esta ponderação de termos foi proposta inicialmente por [Luhn \(1957\)](#). Outro exemplo é a observação de que se um termo aparece em muitos documentos, portanto é muito comum, estes devem sofrer penalização pois tem baixa especificidade de documentos relacionados. Esta técnica é chamada de *Inverse Document Frequency* (IDF), e foi concebida por [Jones \(1972\)](#), por meio de análises estatísticas. A equação 2.1 define a ponderação TF :

$$TF_{i,j} = \begin{cases} (1 + \log f_{i,j}) & \text{se } f_{i,j} > 0 \\ 0 & \text{caso contrário} \end{cases} \quad (2.1)$$

onde, i denota o termo, j o documento onde o termo se encontra e $f_{i,j}$ é a frequência do termo i no documento j . A equação 2.2, por sua vez define a ponderação IDF :

$$IDF_i = \log \frac{N}{n_i} \quad (2.2)$$

onde i identifica o termo no vocabulário geral da coleção, n_i remete ao número de documentos que o termo i aparece e N é o total de documentos na coleção.

As duas ponderações de termos citadas acima foram combinadas por [SALTON e YANG \(1973\)](#), para formar a ponderação *term frequency - inverse document frequency*

(TF-IDF). Esta abordagem é interessante, pois visa utilizar os pontos fortes de ambos pesos e, simultaneamente, diminuir os pontos fracos de ambos. A Fórmula 2.3 define o cálculo da ponderação TF-IDF:

$$w_{i,j} = \begin{cases} (1 + \log f_{i,j}) \times \log \frac{N}{n_i} & \text{se } f_{i,j} > 0 \\ 0 & \text{caso contrário} \end{cases} \quad (2.3)$$

onde o peso TF-IDF, $w_{i,j}$ do termo i no documento j é encontrado utilizando a frequência do termo no documento, denotada por $f_{i,j}$, o número total de documentos na coleção, N , e o número de documentos em que o termo i aparece na coleção, n_i .

O modelo vetorial, que modela a similaridade de consultas e documentos como um vetor, utiliza a ponderação TF-IDF para compor sua função de ranqueamento. Este modelo tornou-se popular devido à sua eficácia, sendo utilizado também em áreas correlatas. Em, Ko (2012) e Lan et al. (2005) o modelo vetorial e a ponderação TF-IDF foram utilizados como base para criação de técnicas melhores para classificação de textos em categorias e em Ramos (2003) para identificar palavras que seriam melhores candidatas para compôr uma consulta.

2.1.6 Recuperação

A última fase do procedimento de RI consiste em criar uma listagem contendo o número real, representando a similaridade entre a consulta e um documento, e o documento em si e ordená-la de forma decrescente. O resultado é o *ranking*, ou classificação, dos documentos mais relevantes para a consulta submetida.

Esta lista deverá ser exibida ao usuário que submeteu a pesquisa. A mesma será inspecionada pelo utilizador do sistema e os documentos serão requisitados de acordo com alguma informação que normalmente é fornecida juntamente da listagem, como por exemplo: título, descrição, *snippet* (fragmento do conteúdo), entre outros.

2.2 Sistemas Distribuídos

Inicialmente, os sistemas eram centralizados e consistiam em uma única grande máquina que ocupava uma sala inteira, como o ENIAC (ECKERT; MAUCHLY, 1964), um dos primeiros computadores eletrônico-digitais inventados. Ao longo do desenvolvimento da computação, os computadores diminuíram de tamanho e melhoraram sua capacidade de processamento e memória.

Os primeiros sistemas distribuídos começaram a surgir com a invenção dos mini computadores, por volta da década de 1970 (WESTERINEN; BUMPUS, 2003). Na década de 1980, os computadores pessoais incrementaram a tendência já iniciada de distribuição

de sistemas. Quando as redes locais foram criadas, a nova tecnologia serviu de base para o compartilhamento de hardware e software. Neste ponto, a indústria já havia começado uma mudança de paradigma, onde antes haviam aplicações centrais localizadas em grandes *mainframes*, agora permeava a ideia de aplicações cliente-servidor.

Como proposto por Fox (1981), sistemas distribuídos podem ser analisados como uma organização ou empresa. A menor organização possível é a de uma única pessoa, que assume o papel de um processo. Nesta configuração, todas as tarefas são geridas e executadas por esta pessoa. A capacidade da organização ou sistema é limitada, portanto, à capacidade de processamento e recursos do indivíduo.

Para ser capaz de processar um volume maior de dados, a organização deve crescer. Uma organização com mais pessoas, no entanto, implicará em aumento de complexidade de controle. Quando um grupo de pessoas estiverem trabalhando, deverá haver coordenação para a execução das tarefas. Da mesma forma, um sistema distribuído, uma vez que possua várias unidades que realizam processamento ou armazenamento, possuirá um nível de complexidade maior devido à necessidade maior de controle e coordenação. Em troca, haverá aumento nas capacidades de processamento e armazenamento.

Como citado anteriormente, sistemas de RI atuais como o (Google, 2018), são implementados de maneira distribuída. A distribuição do sistema pode variar de uma arquitetura cliente-servidor, com somente um servidor, até uma rede *peer-to-peer* com vários servidores. A distribuição do sistema de RI não é essencial para o processo de RI em si. No entanto, dada a massa de dados que os sistemas atuais recebem de entrada e a quantidade de requisições que os mesmos devem atender, um sistema centralizado torna-se, rapidamente, inviável.

Neste trabalho, é desenvolvida uma rede *peer-to-peer* de servidores, onde pode haver mais de uma instância de servidor por máquina. O propósito desta arquitetura é fornecer funcionalidades de sistemas distribuídos, como: tolerância a falhas, alto desempenho, redundância, escalabilidade, entre outros, ao sistema de RI.

2.3 Frameworks

Inicialmente, os programas executavam tarefas simples e possuíam poucas linhas de código se comparados aos atuais. Isto ocorria devido as várias limitações da época: processadores lentos, pouca memória de persistência de dados, pouca memória temporária, linguagens de programação simples, entre outras. Conforme os *softwares* eram desenvolvidos, as mais variadas áreas começaram a ver a possibilidade de utilização da computação para melhorar ou facilitar seus processos. Processamento de folhas de pagamento, automatização de produção, contabilidade, são exemplos desta utilização fora do campo de origem.

Com isso, novas aplicações precisaram ser implementadas e estas tornavam-se cada vez mais complexas. Para atender à crescente demanda, os programadores começaram a perceber que não era sustentável, ou prático, escrever novamente códigos de funcionalidades já desenvolvidas. Juntamente com a percepção de que, via abstração do código, era possível reutilizá-lo em várias aplicações, as primeiras bibliotecas começaram a aparecer. Inicialmente, consistiam de apenas alguns artefatos de *software*, em sua maioria funções e estruturas de dados, e possuíam um caráter muito pessoal, pois eram criadas e mantidas de forma individual pelos programadores.

Estas bibliotecas pessoais foram o início do que hoje são conhecidos como *frameworks*. Estes, são grupos de classes, estruturas de dados, implementações prontas para serem utilizadas e abstrações, que operam em conjunto para prover algum tipo de funcionalidade de altíssimo nível. Em [Roock, Wolf e Züllighoven \(1998\)](#), *frameworks* são definidos como um conjunto de conceitos que colaboram entre si, feitos para reutilização. O autor chega a esta definição pois esta é capaz de cobrir os diferentes tipos de *frameworks*: conceituais, arquiteturais e de software.

A realidade do universo de *frameworks* da computação atual, consiste em vários tipos de abstrações e implementações. Existem *frameworks* para arquitetura do software ([ZACHMAN, 1987](#)), para a criação de interface gráfica ([jQueryUI, 2018](#)), para funcionalidades específicas, como autenticação ([OAuth, 2012](#)) ou utilização de formatos de arquivos complexos como *Portable Document Format* (PDF) ([iText, 2018](#)), entre outros.

De acordo com o sugerido por [Fayad e Schmidt \(1997\)](#), há um alto custo na indústria de software devido ao não reaproveitamento de conceitos e componentes. Portanto, é possível afirmar que o principal conceito que fundamenta o uso dos *frameworks* é o da reusabilidade de artefatos de software que comprovadamente funcionam. Dessa forma, o principal benefício de sua utilização se manifesta na economia, tanto de tempo, quanto de dinheiro. Como consequência da economia de tempo causada pelos *frameworks*, a velocidade de desenvolvimento do *software* aumenta, tornando mais próximos a concepção do programa e sua primeira versão *alpha*.

2.4 Trabalhos Correlatos

Esta seção apresenta trabalhos similares à proposta desta monografia de um *framework* para criação de sistemas de recuperação de informação distribuídos.

2.4.1 Apache Lucene

O Apache Lucene consiste em um *framework open-source* para organização e recuperação da informação, desenvolvido em Java. O objetivo dele é fornecer alto desempenho concomitantemente com interfaces para criação de consultas que retornem documentos

relevantes. Dentre os pontos fortes do Lucene, pode-se citar: escalabilidade, performance, portabilidade entre sistemas, implementações em outras linguagens, algoritmos prontos para busca na coleção, rápida indexação. Embora o Lucene ofereça modelos de ranqueamento prontos, também é possível implementar modelos personalizados por meio das interfaces fornecidas. (Apache Lucene, 2017).

O projeto surgiu como uma forma de aprender a linguagem Java, tendo Doug Cutting como criador, em 1997. De acordo com Białecki, Muri e Ingersoll (2012), o Apache Lucene pode ser dividido em quatro categorias, a saber: Análise de Linguagem, Indexação e Armazenagem, Consultas, Opções Acessórias.

A análise de linguagem é a responsável no sistema por realizar o pré-tratamento, tanto nos documentos de entrada quanto nas consultas, com o objetivo de transformar os dados para o formato utilizado pelo modelo lógico interno do *framework*. Nesta parte também ocorre a criação de *tokens* que serão inseridos no índice invertido, no caso dos documentos, ou utilizados para criar uma representação que o sistema compreenda, no caso das consultas.

A categoria de Indexação e Armazenagem do Lucene é composta por uma série de características interessantes para um sistema de RI distribuído. O processo de indexação é quase em tempo real, o que possibilita um documento recém adicionado estar disponível para consulta quase que imediatamente. Os índices podem ser criados de forma segmentada, como ocorre em sistemas distribuídos, com políticas para realizar a união destes fragmentos. Oferece suporte a transações como em bancos de dados relacionais, o que fornece consistência dos dados e resistência a falhas. Existem também abstrações que permitem ao usuário utilizar diferentes estratégias para I/O (*Input/Output*), armazenagem e estruturas de dados.

Na categoria de consultas, Lucene se destaca por oferecer várias representações de consultas prontas, além de um *framework* para o desenvolvimento de uma representação personalizada. Existe a opção de utilizar um *pseudo-feedback* de relevância, o que permite melhoria da RI do sistema. A partir da versão 4 do Lucene, é possível também acoplar modelos de ranqueamento próprios, desenvolvidos com base nas interfaces do *framework*. No entanto, o sistema já possui implementações de vários modelos de ranqueamento amplamente utilizados como o modelo vetorial ou o Okapi BM25 (ROBERTSON; ZARAGOZA, 2009).

Outra característica interessante do Lucene é a utilização de vários módulos para oferecer recursos que incrementam a recuperação dos documentos também conhecido como opções acessórias. Um exemplo seria o *Highlight*, ou marcação, dos resultados, no qual as palavras do documento que coincidem com as fornecidas na consulta são iluminadas ou marcadas com uma cor de fundo diferente para destaque. Outro módulo interessante é o de agrupamento de documentos por meio de uma chave, como por exemplo, documentos

que estão sob o mesmo URL (Uniform Resource Locator).

2.4.2 Apache Solr

Apache Solr é uma implementação e extensão do Apache Lucene moldada como um servidor de RI de alta performance e escalabilidade. Solr utiliza outra tecnologia da Apache, o ([Zookeeper](#), 2008), para tornar a aplicação escalável, distribuída, tolerante à falhas e replicada.

Além disso, mantém as outras características já expostas do Lucene e acrescenta várias outras funcionalidades relevantes ao ambiente comercial como: opções de estatística e monitoramento, segurança, armazenagem avançada, escaneamento de documentos do tipo *Rich-Text* como PDFs e arquivos do Microsoft Word, várias opções de protocolo de comunicação e dados, entre outras ([Solr Features](#), 2017).

2.4.3 Vespa

O Vespa é o equivalente do Lucene ou Solr, desenvolvido pela Yahoo. A proposta do Vespa é prover computação sobre grandes coleções de dados em dezenas de milissegundos ([Yahoo Vespa Docs](#), 2017). O objetivo do sistema é utilizar computação distribuída para obter um tempo de consulta quase constante independente da quantidade de documentos ou dados em que é realizada a busca. Para tal, são utilizadas, além da distribuição, replicação da informação entre servidores e consultas distribuídas que ocorrem paralela e simultaneamente.

O controle de todo o sistema é centralizado no chamado *Application Package*, porém, configurações individuais de cada máquina pode ser realizadas para um processo de ajuste fino da aplicação. Também existe a possibilidade de acoplamento de extensões em Java, com o intuito de mudar aplicar comportamento personalizado, seja na parte de consultas, resultados ou escritas que o sistema realiza.

2.4.4 Carrot²

Carrot² é um *framework open-source*, desenvolvido no instituto de ciência da computação da Universidade de Tecnologia Poznan na Polónia, por [Osiński e Weiss](#) (2005). O objetivo era implementar um *framework* para recuperação da informação e *data mining* na *web*, que oferecesse pesquisas rápidas. As principais metas almejadas eram eficiência no processamento e flexibilidade.

A flexibilidade do sistema é devida a uma primeira parte da arquitetura baseada em XML. Como o XML é um tipo de arquivo fácil de ser lido tanto por humanos quanto por computadores, esta arquitetura acrescenta facilidade e usabilidade para o *framework*. A segunda arquitetura presente no *Carrot²* é denominada pelos autores de arquitetura de

interfaces locais. Esta, por sua vez, é responsável por prover o requisito de eficiência de processamento.

*Carrot*² contém implementações de técnicas de RI como *stemming* e filtro de *stopwords*. Além disso, vários algoritmos para criação de *clusters* de resultados estão contidos no pacote. Alguns deles são: K-means, *Suffix Tree Clustering (STC)*, Lingo e *Fuzzy Clustering*.

Com o objetivo de atender às necessidades tanto de ambientes comerciais quanto acadêmicos, o *Carrot*² foi desenvolvido com base nos seguintes requisitos: Arquitetura de Componentes, onde subentende-se que o formato do projeto deveria ser o de um *framework*; Flexibilidade, para permitir que os componentes fossem fáceis de usar, autônomos e personalizáveis; Portabilidade, definida como independência de Sistema Operacional (SO) e linguagem de programação; Alto desempenho, pela questão da eficiência e Opções de Licença Permissiva, para permitir o uso indiscriminado do *software* tanto para fins comerciais quanto pessoais.

2.4.5 *The Information Grid*

O projeto *The Information Grid* trata de um *framework* para implementação de aplicações orientadas a objetos, que sejam principalmente de RI ou ferramentas pessoais que necessitem realizar RI para documentos e apresentá-los utilizando uma interface gráfica. O grande diferencial do trabalho com relação aos anteriores é a utilização de um modelo de interação para auxiliar os usuário a encontrar os documentos de que precisam, (RAO et al., 1992).

Baseado no paradigma de recuperação de objetos de forma iterativa e associativa, o *software* cria uma interface gráfica amigável e de rápida aprendizagem. A iteratividade do paradigma encontra-se na utilização do programa. O usuário recupera documentos realizando vários ciclos de consultas, onde a técnica de *feedback* de relevância é aplicada. Dessa forma, as consultas subsequentes tornam-se cada vez mais precisas, baseado no que foi criticado em consultas anteriores.

A justificativa para esta abordagem iterativa recai sobre a característica do domínio do problema para o qual a aplicação foi imaginada. Em certas situações de RI, os usuários não sabem exatamente que documentos estão disponíveis ou para quais diferenças deve se ater a fim de encontrar o que procura. Neste ponto, assemelha-se à recuperação na web. No ambiente corporativo também é possível encontrar situações deste tipo onde a coleção pode ser muito diversa e o usuário não tem ciência exata dos documentos que necessita

2.4.6 Comparação

A proposta deste trabalho se assemelha mais com os trabalhos ([Apache Lucene, 2017](#)), ([Solr Features, 2017](#)) e ([Yahoo Vespa Docs, 2017](#)), principalmente. Estes *frameworks* são todos destinados à criação de um servidor ou uma rede de servidores de RI distribuída. Alguns permitem a implementação de componentes que podem ser acoplados, como é o caso do ([Apache Lucene, 2017](#)). No entanto, a diferenciação entre a proposta deste projeto com relação aos demais, recai principalmente na capacidade de implementação por parte do usuário, de qualquer componente do sistema que possua relevância para a RI.

Inclusive, é possível a substituição destes componentes em tempo de execução. A facilidade para o acoplamento também é um diferencial. Mesmo o *framework* sendo utilizado por programadores inexperientes, é possível que estes desenvolvam e utilizem seus próprios componentes. A flexibilidade almejada também permite a utilização para pesquisa em RI e áreas correlatas.

3 Desenvolvimento

Neste capítulo, inicialmente haverá a apresentação da arquitetura utilizada para o desenvolvimento do *framework* e sua divisão em duas camadas. Posteriormente serão discutidas as interfaces da camada lógica que compõem o *framework*, detalhando seus papéis individuais e suas interações. Em seguida, a camada de comunicação será exposta, com foco na utilização do *Thrift* e da implementação do *Chord* dentro do *framework*, bem como as interfaces e componentes que são responsáveis pela intercomunicação das camadas.

3.1 Arquitetura do *Framework*

A arquitetura do *framework* pode ser dividida em duas camadas: uma camada lógica e uma camada de comunicação. Em linhas gerais a camada lógica é a responsável pelo trabalho de RI. Portanto, contém as interfaces e as implementações dos modelos lógicos inerentes à organização e recuperação da informação. A segunda camada, de comunicação, compreende o código necessário para a comunicação cliente-servidor e servidor-servidor, e que fornece as funcionalidades criação de servidores do *DORIF*. A Figura 3, ilustra as etapas e componentes envolvidos em uma comunicação entre cliente e servidor. Há também na ilustração a divisão das camadas internas do *framework*.

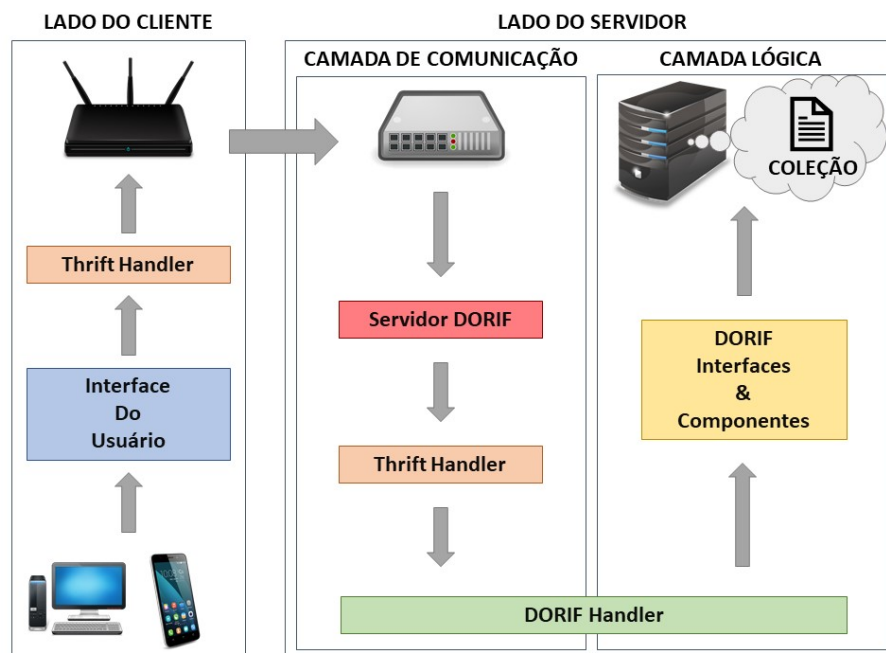


Figura 3 – Diagrama de alto nível representando a arquitetura entre cliente e servidor.

3.1.1 Camada Lógica

A camada lógica contempla a parte conceitual do *framework*, ou seja, toda a abstração de alto nível reside nas interfaces definidas nesta camada. Para fornecer a flexibilidade almejada, foram projetadas treze interfaces. Por definição, estas representam o contrato que determinados tipos de objetos do sistema implementam e que permite sua comunicação, cooperação e fraco acoplamento.

O maior desafio dentro do escopo do problema de um sistema de RI é como os vários sub-módulos, ou sub-sistemas, se comunicam e compartilham as informações. As tarefas e algoritmos que estes componentes executam são, a princípio, relativamente simples, deixando a complexidade localizada nas interconexões. Um exemplo prático que comprova este fato, é a etapa de cálculo da similaridade. Trata-se de um simples cálculo, algo trivial para um computador, porém, que requer os termos e seus respectivos pesos, de um documento, de uma consulta e do vocabulário global da coleção.

No total, onze interfaces compõem a camada lógica do *DORIF*. Nas próximas subseções cada uma delas é discutida. Antes, no entanto, justifica-se a escolha de interfaces para programação da parte lógica. Em sequência a *ICollection Handler* é a primeira a ser abordada pois trata-se do centro do sistema de RI do *framework*. É responsável por coordenar os demais componentes. Posteriormente são explicadas algumas interfaces de base: A *IFactory*, uma fábrica para instanciamento de objetos do sistema de maneira delegada; *ICollector*, que possui o trabalho de encontrar e coletar os documentos; *IPre Processor*, realiza um tratamento de normalização de texto nos documentos e consultas; *ITerm*, define o modelo lógico de termo para o sistema e *IWeight*, especifica a lógica de um determinado peso de ponderação.

Após as interfaces mais básicas, são expostas as de nível mais alto, que utilizam as de base. São elas: *IDocument*, consiste no modelo lógico de documento, *IVocabulary* representa o modelo lógico de vocabulário; *IQuery* define o modelo lógico da consulta e *IRankingFunction*, responsável pela função de ranqueamento utilizada para o cálculo de similaridade. Todas as implementações citadas a seguir podem ser encontradas em (DORIF..., 2018).

3.1.1.1 Interfaces

As interfaces foram utilizadas para permitir que os componentes pudessem compartilhar as informações que necessitam para a execução de suas respectivas funcionalidades. Até o momento, o termo interface foi utilizado no sentido amplo dentro da teoria de orientação a objetos. No entanto, é possível implementá-las na linguagem Java utilizando dois recursos: classes abstratas ou interfaces.

No caso, optou-se pela utilização de interfaces em detrimento de classes abstratas,

pois as últimas requerem que as classes que utilizarão seus recursos utilizem herança. Em Java não existe a possibilidade de herança múltipla, logo, caso fossem utilizadas classes abstratas não seria possível que os componentes desenvolvidos com o *DORIF* herdassem características de outras classes dos projetos dos usuários.

Por outro lado, teoricamente não existe limite do número de implementações de interfaces em Java. Dessa forma, um usuário do *framework* pode criar uma classe que: possua herança com outra classe de seu sistema, mantenha as funcionalidades estipuladas pelas interfaces do *DORIF* e faça parte do sistema de RI personalizado.

3.1.1.2 *Collection Handler*

A arquitetura do *DORIF*, como explicado na seção anterior, é baseada em interfaces que permitem os componentes se comunicarem de maneira modular. Durante a construção do *framework* alguns padrões de projeto foram utilizados como o *Observer* e o *Factory Method*. De acordo com [Gamma et al. \(1994\)](#), a utilização de padrões de projeto contribui para a modularidade e reusabilidade do código. Futuras modificações, expansões e manutenções também são facilitadas devido a utilização destas técnicas.

O centro da arquitetura do *DORIF* reside em um componente chamado *Collection Handler*. Trata-se do responsável por intermediar a camada lógica com a camada de comunicação, servindo como um tipo de *proxy* entre as duas. As requisições chegam por intermédio do servidor *Thrift*, utilizando o *DORIF Handler* que será explicado posteriormente, e o mesmo realiza chamadas para métodos contidos no *Collection Handler*. O primeiro método a ser executado deverá ser o *init()*. Dentro dele deverá ser colocada a lógica para que o *Collection Handler*, implementado pelo usuário, funcione corretamente. Normalmente, as atribuições de componentes que compõem o *Collection Handler*, devem ser feitas no construtor e quaisquer preparativos destes componentes devem ser feitos no *init()*.

Sumarizando, o papel principal do *Collection Handler* consiste em fornecer uma única interface de acesso RI para o meio externo e, internamente, coordenar os diferentes componentes que fazem parte do sistema de modo que atuem de forma coesa, porém, modularizado e encapsulado. Para isso, age como ponte solicitando ao meio externo o que o meio interno requisita para operar, e utilizando o meio interno para prover funcionalidades consumidas pelo externo. Além disso, executa algumas funções que podem ser classificadas como de manutenção do sistema, são elas: a atualização da coleção de documentos, que são fornecidos pelo coletor de documentos e armazenados no *Handler* e atualiza os pesos dos documentos armazenados de tempos em tempos.

O *Collection Handler* também é o responsável por receber uma consulta e retornar o *ranking* de documentos relevantes à aquela consulta. No entanto, para diminuir o tempo de resposta entre servidor e cliente, o ordenamento em ordem decrescente dos

documentos somente é realizado no servidor que está diretamente conectado com o cliente que iniciou a consulta. Evitando realizar ordenações parciais nos servidores, acelera-se a resposta. A Figura 4 ilustra os relacionamentos dos meios internos e externos para com o *CollectionHandler*.

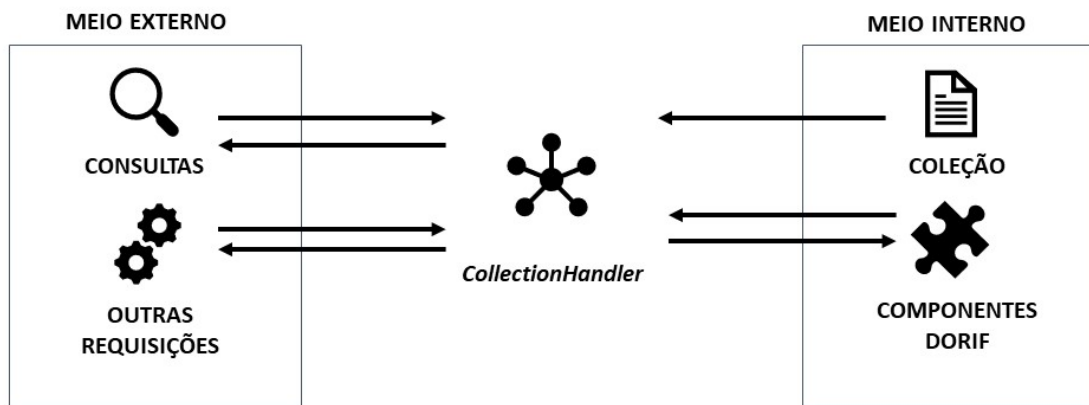


Figura 4 – Diagrama representando o papel do *CollectionHandler* dentro do *framework*.

3.1.1.3 Factory

O primeiro componente que deve ser inserido em um *Collection Handler* é uma *Factory* que implementa a interface *IFactory* do *DORIF*. É por meio desta que serão criados os termos globais e de documentos, além das consultas. Esta escolha arquitetural foi feita baseada no padrão *Factory Method*. Ele consiste basicamente em se definir, em uma *superclasse* ou interface, a assinatura de uma operação de criação de um objeto e, deixar para as classes filhas ou que implementam a interface, a responsabilidade de implementar as operações. É um padrão utilizado em *frameworks* arquiteturais por facilitar a personalização da implementação, sem atribuir muita complexidade.

3.1.1.4 Collector

O próximo componente fundamental para o funcionamento do *Collection Handler* é o *Collector* que implementa a interface *ICollector* do *DORIF*. Este é o responsável

por, dada uma fonte de documentos, coletá-los e notificar o *Collection Handler* que novos documentos estão disponíveis para indexação. A notificação é realizada por meio da implementação do padrão de projeto *Observer* (GAMMA et al., 1994). Neste caso, não existe uma interface ou classe abstrata própria para a utilização do padrão. Os próprios *Collection Handlers* podem se cadastrar em *Collectors* e ambos já possuem métodos de notificação e atualização, além de cadastro e descadastro, previstos no *Observer*.

Considerando-se que a coleta de documentos deverá ser contínua, uma implementação de *ICollector* realizará um ciclo como o demonstrado na Figura 5. A *thread* em que o coletor executa observará algum tipo de diretório ou fonte de documentos. Uma vez que novos documentos forem encontrados a coleta dos mesmos será realizada, criando os modelos lógicos de documentos do sistema. Finalmente, uma notificação será feita para o *Collection Handler*, o que se traduz no método de notificação do padrão *Observer*.

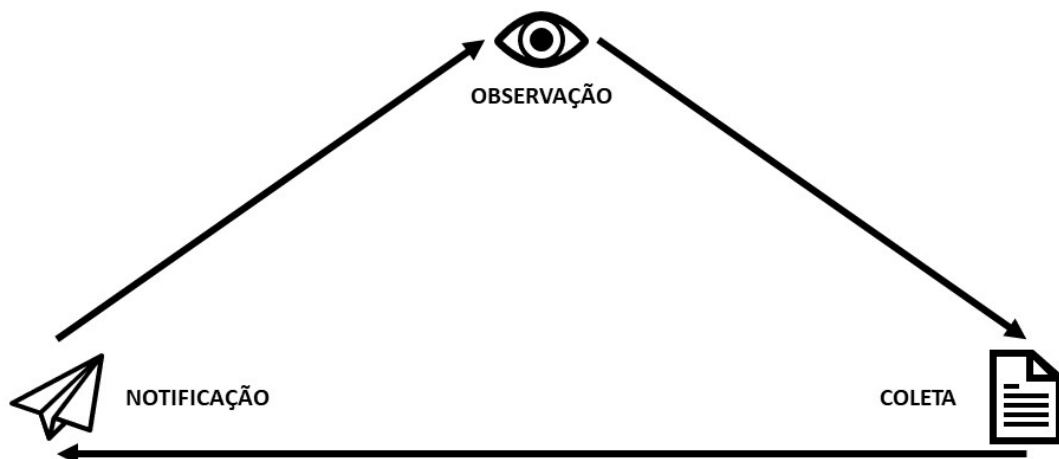


Figura 5 – Ilustração do ciclo executado por um *Collector*.

3.1.1.5 Pre Processor

Durante as etapas de RI, tanto a indexação dos termos em documentos e em consultas, necessitam de que um pré-processamento do texto seja realizado. Este processo garante uma normalização do texto e também permite diminuir diferenças que são irrelevantes para a recuperação da informação, como letras maiúsculas, artigos, acentos, entre

outros. Para realizar esta função, o *DORIF* disponibiliza uma interface chamada *IPreProcessor*. A implementação desta representa a criação de um módulo do sistema, que será posteriormente acoplado no *Collection Handler* e nas consultas. No segundo caso, a *Factory* realizará o acoplamento durante a criação do objeto de consulta.

3.1.1.6 *ITerm*

O módulo que implementa a interface *IPreProcessor*, após o processamento de determinada entrada seja um documento ou uma *String*, criará uma lista de objetos que implementam a interface *ITerm*. Estes são o modelo abstrato dos termos indexados. A modularidade fornecida pela interface, permite a adição, utilização e atualização de uma quantidade variável de pesos. Desta forma, é possível para uma implementação do *DORIF* criar um tipo de termo global e um tipo local, do documento, cada um com seus respectivos pesos.

3.1.1.7 *IWeight*

Cada peso, presente em um objeto do tipo *ITerm*, deve implementar a interface *IWeight*. Por meio desta, é possível abstrair os cálculos dos pesos e suas respectivas atualizações, bem como a adição de variáveis específicas. Neste último caso, os atributos próprios devem ser controlados posteriormente por meio de *casts* e de acordo com a lógica específica da aplicação que estiver utilizando o *DORIF*.

Por padrão, a interface *IWeight* fornece suporte à pesos baseados em frequência, uma vez que são comuns entre modelos, como o Booleano, o Vetorial e o Probabilístico. Cada um utiliza o número de repetições de um termo de uma maneira específica. No modelo Booleano, por exemplo, é utilizada apenas para saber se o termo está presente ou não. No Vetorial e no Probabilístico, a frequência é utilizada como base nos cálculos de pesos e probabilidades respectivamente.

3.1.1.8 *IDocument*

Durante a etapa de coleta de documentos, realizada pelo componente *Collector*, os objetos gerados implementam a interface *IDocument*. Ela define um conjunto básico de métodos que todo documento deve fornecer, independentemente de seu formato, para ser um modelo lógico apto a operar no sistema. A lógica que permeará os métodos da interface deve ser implementada pelo usuário. Uma vez pronta, o modelo lógico de documento é abstrato o suficiente para não ser dependente de formato, ou tipo, e quando for solicitado para entrega em um cliente, o mesmo pode ser lido do disco de um servidor e enviado.

O *DORIF*, em seu modelo lógico de documento, oferece suporte nativo à normalização do documento, pois é uma boa prática dentro da área de RI, uma vez que quando normalizado, o tamanho do documento não interfere em pesos como o TF. Caso contrário,

um documento grande hipoteticamente possuiria mais termos repetidos, o que ocasionaria em um TF de valor maior. Em contrapartida, um documento menor não seria composto por tantos termos repetidos o que levaria a um TF menor. Ao utilizar a normalização do documento como divisor no peso, os documentos grandes são penalizados pelo seu tamanho, tendo como consequência um TF menos enviesado.

3.1.1.9 *IVocabulary*

O controle dos termos, tanto localmente nos documentos quanto no vocabulário compartilhado (global), é realizado por meio de objetos que implementam a interface *IVocabulary*. Embora seja simples, possuindo apenas quatro métodos, a abstração proporcionada pela interface permite a utilização em ambos os casos anteriormente citados, pelos mesmos métodos. Isto agrega facilidade durante a utilização devido à simplicidade.

Ao mesmo tempo, é possível implementar qualquer lógica que seja necessária para realizar o controle dos termos no vocabulário, desde que a interface seja atendida. Implementações da *IVocabulary* do *DORIF*, podem ser desde estruturas de dados prontas, disponíveis de maneira nativa em linguagens orientadas a objetos, até estruturas complexas criadas para otimização de propósitos específicos.

Na Figura 6 é demonstrada como é realizada a composição entre as interfaces do sistema com relação à documentos, vocabulários, termos e pesos. Um documento possui um vocabulário. Este por sua vez comporta todos os termos indexados daquele documento. Finalmente, cada termo possui suas instâncias de pesos dentro do conjunto de tipos de peso definidos pelo modelo de RI.

3.1.1.10 *IQuery*

Uma vez que as interfaces anteriores já estão implementadas em componentes no sistema e os mesmo estão conectados pelas interfaces, um novo objeto, que conclui o ciclo do sistema pode ser instanciado. Este objeto consiste em um componente que implementa a interface *IQuery*, do *DORIF*, e representa a consulta a ser submetida no sistema de RI.

A consulta é submetida ao mesmo *Pre Processor* que os documentos. A definição de um *Pre Processor* nas consultas é feita de forma automática pela *Factory* que é encarregada de criar, além dos termos, as consultas do *DORIF*. Além disso, há uma conexão com o vocabulário global, uma vez que pesos como o TF-IDF necessitam de ambos vocabulários locais dos documentos e globais da coleção.

Fazem parte da interface, os métodos: *prepare()* e *isPrepared()*, ambos servem para facilitar a implementação em sistemas distribuídos. Nestes sistemas, como os fluxos de execução dependem de tarefas que contatam outros servidores, muitas vezes os componentes precisam se comunicar de tal forma a ter ciência se os outros estão prontos ou não para

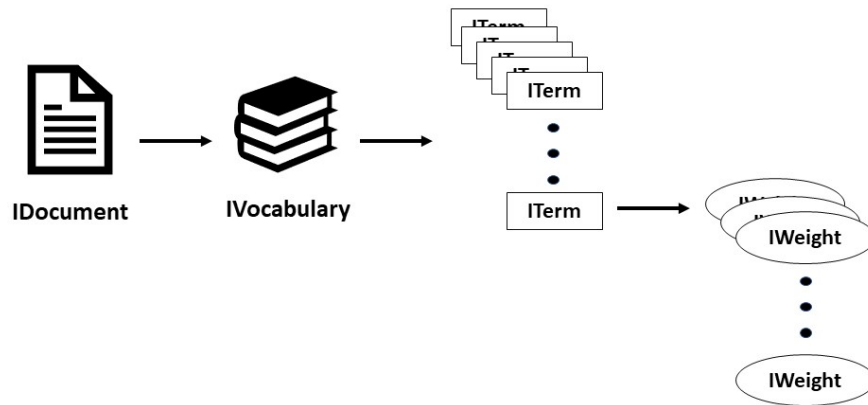


Figura 6 – Esquema de composição de documentos, vocabulários, termos e pesos.

executar determinada tarefa.

A consulta passa por um processo de preparo, iniciado pelo usuário, onde seu vocabulário interno e a computação de seus pesos é realizada. Uma vez preparada, a mesma é submetida ao *ICollectionHandler* para que seja testada contra os documentos da coleção, levando à recuperação final.

3.1.1.11 *IRankingFunction*

Após a submissão de uma consulta, um objeto que implementa a interface *IRankingFunction*, fornecido ao *CollectionManager*, é o responsável por comparar uma *Query*, ou consulta, com um *Document*, ou documento. Em RI, este papel é desempenhado por uma *Função de Ranqueamento*. O objetivo final é retornar um valor real que represente o grau de similaridade entre uma consulta em um documento de uma coleção.

A utilização da função de ranqueamento é a última etapa necessária à recuperação de informação. No final, após comparar a consulta com todos os documentos da coleção, por intermédio da função, obtém-se uma lista de valores reais de similaridade e seus respectivos documentos. O último passo antes de retornar com a resposta à consulta para o usuário, é ordenar a lista por ordem decrescente de similaridade, ou seja, os documentos com maior similaridade, portanto, mais relevantes, estarão no topo da lista. Consequentemente os de menor relevância ocuparão posições inferiores na lista, inclusive podendo não ser incluídos a partir de um determinado limiar ou se seus valores de similaridade forem zero, por exemplo.

3.1.2 Camada de Comunicação

Nesta subseção será discutida a camada de comunicação do *framework*. Na seção anterior a camada lógica foi exposta, no entanto, nela não havia nenhum componente ou interface que fosse encarregado da comunicação ou distribuição do sistema em si. Toda a lógica referente à funcionalidade de servidor, e interações *peer-to-peer*, está contida dentro da camada de comunicação.

O foco desta camada não é fornecer interfaces para criação de servidores altamente personalizáveis. Ainda que haja pontos de customização, esta parte da arquitetura provê todo o comportamento necessário para a criação de um servidor de base para as interfaces do *DORIF*. Além disso, é fornecida também a funcionalidade de intercomunicação entre servidores e entre cliente e servidor.

3.1.2.1 Thrift

Para a implementação de toda a comunicação, foi utilizado o Apache Thrift ([Thrift, 2017](#)). Como explicado anteriormente, este é um *framework* que viabiliza comunicação entre clientes e servidores de diferentes linguagens e tecnologias, por meio de vários tipos de protocolo, com destaque em tipos binários. Devido à grande quantidade de dados com que sistemas distribuídos de RI lidam, a escolha de um protocolo binário torna o tamanho das mensagens e, consequentemente, a transferência de dados, mais rápida e eficiente. Nestes sistemas a resposta rápida é um requisito.

O Thrift mostra-se ideal para o propósito do *framework* devido a seu uso em aplicações como o Lucene ([Apache Lucene, 2017](#)) e o Solr ([Solr Features, 2017](#)), servidores de RI escaláveis; Uber ([UBER, 2018](#)), aplicativo de mobilidade urbana; Evernote ([EVERNOTE, 2018](#)), solução em criação de notas e lembretes; Pinterest ([PINTEREST, 2018](#)), rede social para compartilhamento de ideias; entre outros como pode ser visto em [Powered... \(2018\)](#). Todos são sistemas que demandam respostas rápidas, entre dezenas e centenas de milissegundos, e estão aptos a lidar com quantidades grandes de dados, gigabytes de tráfego de rede mensal.

Embora o *DORIF* possua implementação para servidores Java somente, o Thrift, com sua portabilidade de linguagens, permite que os clientes sejam escritos em várias linguagens de programação amplamente utilizadas como: C#, PHP, Python, Haskell, Node.js, C++, Erlang. Para a utilização em clientes, basta usar o código escrito em *Interface Definition Language* (IDL) e compilado pelo próprio Thrift. Com isto o cliente poderá chamar os métodos providos pelo servidor, que também inclui o código compilado em seus fontes.

Ainda que muito simples, e muito similar à linguagem C, a IDL utilizada no Thrift é muito completa. Com ela cria-se um protocolo para a sua aplicação. Um protocolo

é definido por um conjunto de regras que possibilita a troca de mensagens entre dois sistemas computacionais, e é exatamente o que é feito quando se escreve o código em IDL. Nela podem ser definidos *structs*, que possuem o mesmo papel de classes em linguagens orientadas a objetos; também são definidos *services*, ou serviços, que agem como interfaces Java e possuem a assinatura dos métodos que serão implementados posteriormente no *Handler* do Thrift; existe também a possibilidade de definição de exceções personalizadas que são incluídas nas assinaturas dos métodos nos serviços.

Muitas outras funcionalidades como enumerações, constantes, entre outras, estão presentes na IDL, porém, não serão comentadas neste trabalho. No entanto, é interessante descrever que são previstos tipos de dados específicos para utilização nos códigos de definição de interface do Thrift. Os tipos de dados que se pode utilizar são o conjunto dos tipos básicos nativos, tipos de coleções de dados e dos tipos criados por meio de *structs*, citadas anteriormente.

Quanto aos tipos nativos, são eles: *bool* (tipo booleano), *byte*, *i8* (inteiro de 8 bits), *i16* (inteiro de 16 bits), *i32* (inteiro de 32 bits), *i64* (inteiro de 64 bits), *double* (ponto flutuante), *string* (sequência de caracteres), *binary* (tipo binário para dados não primitivos). Os tipos de coleção de dados são: *list*, uma lista de elementos ordenados; *set*, conjunto não ordenado de elementos únicos e *map*, mapeamento de uma única chave a um único valor.

3.1.2.2 Chord

Para realizar a comunicação cliente-servidor e servidor-servidor, foi utilizado o protocolo *Chord* (STOICA et al., 2001), desenvolvido no MIT como uma solução escalável para buscas por nós (servidores) específicos em redes *peer-to-peer*. Funciona como um mapa que, dado uma chave, um nó é encontrado como responsável por esta chave. É um protocolo escalável pois em teoria e testes, o custo de busca por um nó cresce logaritmicamente de acordo com o número de nós na conectados.

Como consequência do mapeamento realizado é provida indiretamente distribuição de carga entre os servidores conectados, ainda que de maneira pseudo-aleatória. Portanto, não necessariamente existirá uma distribuição igualitária, pelo menos não em pequena escala. À medida em que cresce o número de servidores o problema da distribuição desigual diminui, podendo migrar de alta para baixa, ou até insignificante, desigualdade de chaves.

Para entender o funcionamento do *Chord*, o seguinte processo deve ser explicado. Antes de iniciar a operação dos servidores, é necessário definir um número n de bits que servirá para definir a amplitude das chaves. Uma vez definida esta quantidade, escolhe-se um algoritmo de *hash* criptográfico, como por exemplo MD5, SHA-1 ou SHA-2.

Realizados estes passos iniciais, cria-se a seguinte rotina para geração de chaves.

Dado um novo objeto ou registro para ser incluído no sistema, o mesmo é fornecido como entrada para o algoritmo de *hashing*. O *hash* resultante é convertido em um número inteiro de 64 bits (tipo *long* em Java). Os passos da conversão serão detalhados a seguir.

O resultado do *hashing* do objeto é atribuído a um *array* de *bytes*. Na próxima etapa, é criado um *buffer* de *bytes* e os 64 *bits* mais significantes (da esquerda para a direita) são adicionados ao *buffer*. Em sequência, o *buffer* é invertido para que a leitura seja realizada. Neste ponto os *bytes* presentes no *buffer* são interpretados como um inteiro de 64 bits (*long*). Finalmente, o retorno da função é o resto da divisão inteira entre o inteiro de 64 bits gerado e 2 (dois) elevado ao número de *bits* (o maior valor que pode ser representado com os n *bits* escolhidos).

$$Chave = I_{64} \mod 2^n \quad (3.1)$$

Cada novo servidor no *Chord* gera um *ID*, número de identificação único, utilizando o mesmo processo para geração de chaves e fornecendo seu número de *Internet Protocol* (IP) como entrada para a rotina. É importante esclarecer que o método descrito anteriormente não garante que os números gerados não sejam repetidos. Na realidade, a recomendação dos autores, [Stoica et al. \(2001\)](#) é de que seja escolhido uma quantidade de *bits* grande o suficiente para que seja remota a possibilidade de que dois servidores ou registros recebam a mesma chave.

O objetivo final do protocolo é formar um anel virtual de servidores onde cada um é responsável pelas chaves de número igual ou inferior ao seu ID. A Figura 7 ilustra uma possível distribuição no anel, com três servidores de IDs respectivamente 125, 300 e 689. O ID 0 foi marcado para referência. Neste exemplo, o servidor de ID 689, possuirá o mapeamento para as chaves no intervalo inclusivo de 301 a 689. O nó identificado pelo número 300 se responsabilizará pelas chaves de 126 a 300 e, finalmente, o de ID 125 por chaves de valor 690 até 125, completando o anel. Nota-se que o motivo do termo anel ser empregado se justifica no fato de que uma faixa de chaves pode passar pelo maior número disponível e continuar a partir do zero.

Ao tentar ingressar na rede, os servidores entrantes conectam-se a algum nó já situado no anel e executam uma operação chamada *join* (juntar-se do inglês). O processo consiste basicamente no nó que deseja participar do anel *A* requisitar a este servidor já presente *B*, qual o sucessor de *A*. Este cálculo é realizado por meio do ID de *A* e a busca é realizada utilizando a infraestrutura prevista no *Chord*, detalhada mais à frente. O nó que está conectando então recebe os dados de conexão de seu sucessor e os armazena em estruturas de dados internas. Feito isso, as etapas seguintes do protocolo visam corrigir o anel, atualizando os servidores com o novo participante.

Até o momento na operação de junção, nenhum outro servidor tem ciência da

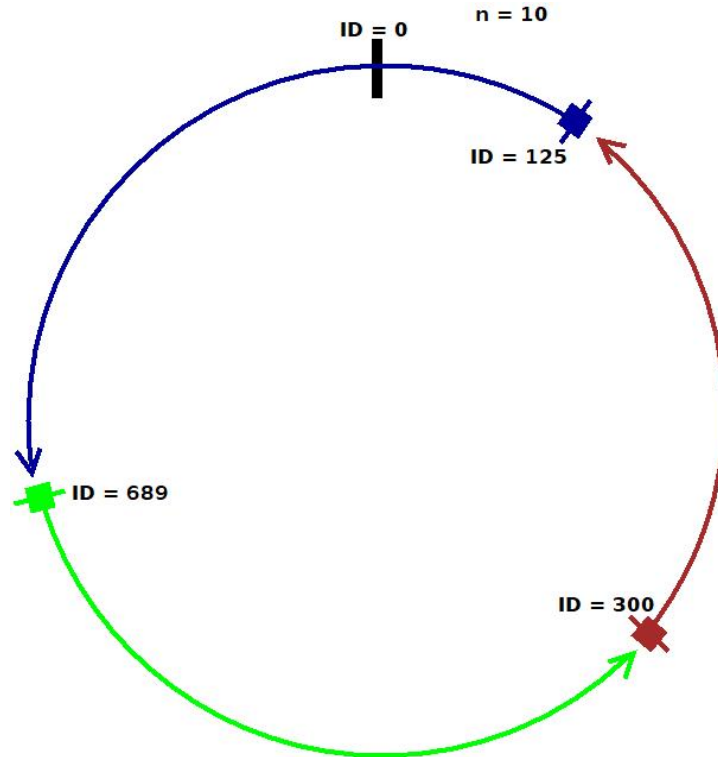


Figura 7 – Diagrama do anel de servidores e suas respectivas faixas de responsabilidade.

presença do mais recentemente conectado. Para que isto ocorra, o primeiro passo é a execução de uma rotina de estabilização no nó mais novo A . Esta rotina verifica se o seu atual sucessor B está correto, requisitando a ele que encontre seu predecessor. Caso o predecessor encontrado C esteja entre o nó solicitante A e seu atual sucessor B , este último é então atualizado no solicitante A com o servidor retornado pela consulta C . Caso contrário nenhuma mudança é feita. Por fim, a estabilização solicita ao sucessor do nó atualizado, neste caso C , ou não, neste caso B , que este execute um método de notificação.

A notificação no sucessor B faz com que este defina seu predecessor como o notificador A , caso o mesmo esteja entre o atual antecessor C e o nó notificado B ou caso a predecessão atual seja nula. Esta rotina é executada automaticamente em uma *thread* independente, de forma a garantir eventual estabilidade do anel. Não há, portanto, esforços para atualizar toda a rede no exato momento de inserção de um novo servidor.

Além da rotina de estabilização, existe em cada nó da rede, uma tabela de apontamentos chamada *Finger Table*. Cada registro contido na tabela possui os dados de um servidor do anel. O nó, cujos dados serão colocados em uma determinada linha da tabela, é o que for encontrado como responsável pelo ID obtido com o seguinte cálculo:

$$ID_i = ID_{servidor} + 2^i \quad (3.2)$$

onde i é o índice da linha da tabela que será preenchida, iniciando-se em zero.

Devido a este cálculo os IDs obtidos crescerão exponencialmente. Esta característica é o que permite às buscas do *Chord* serem realizadas com complexidade $\mathcal{O}(\log n)$ no pior caso, sendo n o número de servidores conectados.

A *Finger Table* é atualizada por fluxo de execução paralelo a cada 10 segundos. Somente uma linha é atualizada por vez. Mesmo que todos os apontamentos tornem-se errôneos, caso os sucessão e predecessão estejam corretos é possível que o sistema continue respondendo às consultas corretamente. No entanto, neste caso, as buscas seria realizadas com uma complexidade de pior caso de $\mathcal{O}(n)$.

Originalmente, o protocolo *Chord* não prevê replicação de servidores. No entanto, é proposto pelo autor no trabalho original, que fossem realizadas cópias dos dados em um determinado número de servidores sucessores. Desta forma caso um nó x pare de responder, os nós $x + 1$, $x + 2$, $x + n$, poderiam, após devida estabilização, responder pelo servidor *offline*. Porém, esta implementação não foi realizada nesta versão inicial do *framework*.

Ainda que não existam réplicas no *DORIF*, o suporte à estabilização pós saída não sinalizada de um servidor, é fornecido. Existem, no entanto, algumas limitações. Não existe a possibilidade de um desligamento informado, ou seja, não há funcionalidade de um servidor informar que se desligará da rede. Outra limitação é que, notificada ou não, a ausência de um nó, mesmo retornando o serviço a um estado estável, não copia os registros do desligado. Não havendo replicação ou cópia dos dados, estes são perdidos uma vez que um nó se ausente.

Para contornar o problema da ausência de nós, poderia ser implementado um esquema de replicação de dados em servidores vizinhos. Como é proposto pelo autor em [Stoica et al. \(2001\)](#), copiando-se o conteúdo de um nó em um determinado número de servidores subsequentes, caso o nó original caia, seus sucessores poderiam suprir seus dados de maneira automática pela própria natureza do protocolo *Chord*.

3.1.2.3 *DORIF Handler*

O *DORIF Handler* é o componente que tem como função realizar a ponte entre a camada lógica e a camada de comunicação. Também serve como *Handler* do Thrift, ou seja, implementa os métodos de comunicação definidos via IDL e, consequentemente, trata as requisições que chegam no servidor. O *DORIF Handler* é implementado pelo *framework*, porém, criado de maneira abstrata de tal forma a não interferir na programação customizada dos demais componentes por parte do usuário.

Esta maneira abstrata de implementação consiste no fato de que o *DORIF Handler* contém um *Collection Handler* e utiliza seus métodos, bem como tem seus próprios métodos utilizados pelo mesmo. No entanto, para o *Collection Handler*, o *DORIF Handler* na

realidade é visto como um *IVocabulary*. Portanto, os métodos utilizados pelo *Collection Handler* são os previstos na interface *IVocabulary*.

Este polimorfismo foi aplicado pois o vocabulário geral precisa consultar outros servidores para conseguir atualizar os termos e seus respectivos pesos. A própria atualização dos pesos, muitas vezes, dependerá de informações inter-servidor, como a quantidade total de documentos no sistema. Tal número, por exemplo, é atualizado periodicamente por uma sub-rotina para manter os pesos eventualmente atualizados.

A eventual atualização dos pesos e informações é algo normal de um sistema de RI. Durante a execução do sistema, espera-se que vários servidores estejam funcionando e, aleatoriamente ou periodicamente, indexando novos documentos. Para garantir que a adição de um novo documento, por exemplo, seja repercutida em todos os nós da rede antes que estes executem uma consulta, haveria um atraso de resposta muito grande. Neste cenário, a perda de acurácia pelos servidores, pouco a pouco se atualizando, não afeta de maneira significativa a recuperação dos documentos.

O prejuízo por não haver atomicidade nas operações consiste em, por exemplo, realizar uma pesquisa em um servidor e não ser recuperado um documento que fora indexado em outro e que estaria incluso no *ranking* caso todos os nós estivessem consistentes com relação aos termos e aos pesos. Uma segunda pesquisa em um momento posterior já poderia incorporar o documento recém adicionado. Portanto, há pouco prejuízo para o usuário e um ganho de resposta e usabilidade que compensa o malefício.

Todos os métodos que realizam operações inter-servidor possuem uma implementação similar. Isto ocorre devido a um cuidado que deve ser tomado para que não ocorram *loops* infinitos ou *deadlocks*. Estes métodos também costumam ser implementados em pares onde um deles simplesmente retorna a resposta baseado nos dados contidos no servidor que executa o método e o outro, além disso, realiza chamadas a um ou mais nós para obter a resposta deles também. O segundo tipo de função é aquela que deve ser chamada pelo cliente, pois consistirá na resposta da rede como um todo. Já o primeiro é chamado por outros nós para atender à requisição do método global.

De maneira geral, o cuidado citado anteriormente se resume em uma verificação se quem deve responder à solicitação é o servidor que está executando a função ou outro nó na rede. Caso seja a primeira opção basta retornar o dado solicitado. Caso seja a segunda, o atual servidor realiza a chamada para aquele que possui o dado, recebe o retorno deste segundo servidor, então retorna a resposta ao solicitante. O *Chord* atua neste ponto facilitando a identificação do responsável pelo registro requerido, por meio do sistema de mapeamento em anel.

A Figura 8 demonstra visualmente a relação entre as interfaces aqui especificadas, assim como a ponte de comunicação entre a camada lógica e a camada de comu-

nicação. Esta é representada pelas relações de agregação entre o *ICollectionHandler* e o *DORIFHandler*. Ressalta-se, entretanto, que o *DORIFHandler* é referenciado dentro do *ICollectionHandler* pela implementação da interface *IVocabulary*. Dessa forma, de maneira indireta e sem conhecimento explícito, há a conexão entre a camada lógica e a camada de comunicação.

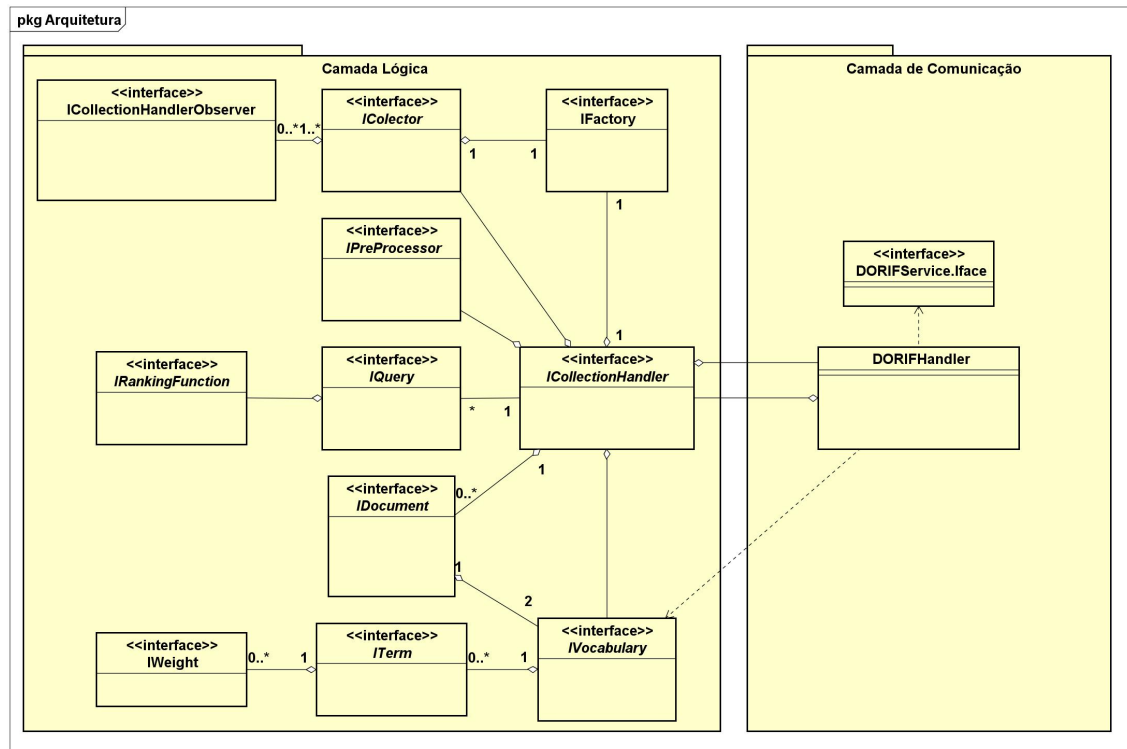


Figura 8 – Diagrama de classes simplificado.

4 Resultados

Neste capítulo serão apresentados os resultados da implementação do *DORIF*. Primeiramente serão apresentados os componentes disponíveis nativamente dentro do *framework*. Será explicado como cada um deles foi codificado e seu papel dentro do sistema. Posteriormente, um estudo de caso contendo um cliente e um servidor será apresentado. Além disso, ao final, o funcionamento do sistema é exposto utilizando os estudos de caso desenvolvidos.

4.1 Implementações Disponíveis

Esta seção contempla algumas decisões de desenvolvimento do projeto, os componentes nativos implementados, bem como seu funcionamento interno no sistema e o processo de construção de um servidor *DORIF* na aplicação do usuário.

4.1.1 Decisões de Projeto

Como é usual em *frameworks*, o *DORIF* oferece implementações prontas. No caso, são oferecidos ao usuário, o código referente à criação e instanciação de um *DORIFServer* bem como uma implementação do modelo vetorial utilizando as interfaces nativas. Para facilitar ainda mais a utilização, foi implementado o padrão de projeto *Builder*. Este padrão é empregado em situações onde a construção do objeto consiste de alguns atributos obrigatórios e outros opcionais, sendo considerada complexa. Visando simplificar, o usuário instancia um *DORIFServerBuilder* passando como argumentos somente uma porta, um tipo de servidor que é definido estaticamente na classe e um *DORIFHandler* implementado pelo utilizador de acordo com suas necessidades. O restante das definições do servidor são definidas via chamada de métodos.

4.1.2 Componentes Nativos

O modelo vetorial fornecido juntamente com o pacote do *framework* contempla classes concretas que implementam as seguintes interfaces: *IWeight*, *ITerm*, *ICollectionHandler*, *IDocument*, *IFactory*, *IQuery*, *IRankingFunction* e *IVocabulary*. Para compor o modelo vetorial foram criadas implementações concretas dos pesos TF, IDF e TF-IDF, comentados no capítulo de revisão bibliográfica. Cada um destes pesos consiste em uma classe implementando a interface *IWeight*. Dentro de cada um, a lógica relativa ao cálculo do peso é definida.

Não necessariamente as implementações concretas possuirão somente os métodos previstos em interfaces. Assim é o caso do peso IDF. Este, por exemplo, necessita de um método próprio que define internamente o número de documentos da coleção. Nestas situações será necessário, em outras classes do sistema, realizar um *cast* para o tipo específico do peso para realizar a chamada do método. O ideal é que tudo pudesse ser realizado dentro do previsto pelo *framework* via interfaces, de maneira abstrata. No entanto, não é possível cobrir todas as possíveis necessidades de cada componente e, mesmo se fosse, haveria uma grande quantidade de métodos inúteis, que seriam utilizados e situações muito específicas somente.

Os pesos citados possuem utilidade apenas se forem vinculados a algum termo. Os termos nativos do modelo vetorial do *DORIF* são o *VectorialDocTerm* e o *VectorialGlobalTerm*. O primeiro é o termo que é utilizado dentro dos documentos em seus vocabulários particulares. O segundo é utilizado para compor o vocabulário global. No termo de documentos, são cadastrados os pesos TF e TF-IDF, pois estes possuem significado vinculado ao termo do documento. No caso do termo global, somente é cadastrado o peso IDF que utiliza a quantidade de documentos na coleção para seu cálculo e, portanto, possui significado no âmbito geral da coleção. A adição dos pesos em cada termo é realizada no método construtor do termo.

A existência dos termos está vinculada aos documentos. Estes por sua vez passam a existir apenas quando são criados a partir de um objeto que implemente a interface *ICollector*. No entanto, não existe nenhuma implementação pronta para uso de coletor de documentos. Isto se deve a dois fatores: o primeiro é que expressa-se como uma interface muito simples, o segundo remete ao fato de que a coleta de documentos é algo muito particular do problema do usuário, portanto, uma implementação pode facilmente não servir de nada para a situação de coleta de um determinado utilizador.

Outra interface que não possui uma implementação concreta nativa e está diretamente relacionada com a coleta de documentos é a *IPreProcessor*. O pré processamento dos documentos possui uma vasta amplitude de maneira de ser realizado. Varia desde processos muito simples de normalização de texto, como mudar todas as letras para minúsculas, até processos complexos como o *stemming*, que buscam o radical do termo. Inclusive existe a possibilidade de diversas combinações destes métodos de tratamento. Logo, deixou-se a cargo do usuário decidir o que e como será usado e implementá-lo, baseando-se na interface *IPreProcessor*, para criar um módulo de pré processamento que será *plugado* em seu sistema.

No entanto, a implementação da interface *ICollector* prevê a utilização de outro módulo do sistema, uma fábrica. Abstraída na interface *IFactory*, uma fábrica no *DORIF* é a responsável por criar termos, documentos e consultas. Nela, serão efetuadas quaisquer lógicas para decisão de que tipo de objeto será instanciado. Dessa forma é possível utilizar

vários tipos de termos, documentos e consultas, dependendo de algum contexto relevante para o problema do usuário.

Quando um coletor de documentos está executando suas rotinas e detecta novos documentos, estes são criados pela fábrica que o usuário definir. Na circunstância da implementação nativa do modelo vetorial, a fábrica instanciará novos *VectorialDocument*. Embora o nome apresente especificidade com relação ao modelo vetorial, este documento não possui qualquer método ou atributo exclusivo para a utilização do modelo vetorial. Os documentos, de maneira geral, não estão acoplados a quaisquer modelos devido à utilização de interfaces para as definições de tipo dos termos. Além dos termos, que estarão em um vocabulário, o documento só contempla informações a respeito de si mesmo, como por exemplo: caminho para posterior carregamento e envio ao cliente, nome, identificador e coeficiente de normalização ou norma.

Como citado anteriormente, os termos dentro de um documento estarão contidos em um vocabulário. Assim ocorre também com os termos globais, que representa o conjunto de termos de toda a coleção. Os vocabulários, no âmbito da implementação do *DORIF* do modelo vetorial, são instâncias da classe *VectorialVocabulary*, que implementa a interface *IVocabulary*. Basicamente, os vocabulários, independente do modelo de RI utilizado, se resumirão a mapas, onde a chave será o texto do termo e o valor armazenado consistirá no objeto que guardará as informações e lógica de pesos deste termo.

Todos estes componentes coexistirão e serão coordenados pelo *VectorialCollectionHandler*. Como uma implementação da interface *ICollectionHandler*, o gerenciador de coleção no modelo vetorial do *DORIF* implementa e executa rotinas de atualização de pesos, realiza o papel de ponte entre as camadas lógica e de comunicação do *framework* e contém toda a indexação dos documentos. É também no *VectorialCollectionHandler* que as consultas serão submetidas.

A criação das consultas, representadas pela classe *VectorialQuery*, é feita externamente, pelo *DORIFHandler*, ao receber a solicitação do cliente. No entanto, a fábrica do sistema é utilizada para a geração do objeto de consulta. Uma vez criado o objeto é submetido ao *VectorialCollectionHandler* para processamento.

A função de ranqueamento do modelo vetorial será requisitada durante a etapa do cálculo de similaridade entre a consulta e um determinado documento. Esta é implementada na classe *VectorialRankingFunction*. Durante a rotina do gerenciador de coleção de processamento de consultas, a função de ranqueamento do sistema é utilizada para calcular o grau de similaridade entre a consulta e cada documento da coleção. Ao final do processo uma lista de graus de similaridade e seus respectivos documentos será retornada.

Dessa forma, todos os módulos do sistema foram utilizados para compor uma implementação do modelo vetorial de RI. Com isso, demonstra-se a aplicabilidade do *framework*

para a implementação de sistemas de RI. São expostas também, as brechas para personalização. A criação de um componente personalizado é obrigatória, no caso do coletor de documentos e pré processador, uma vez que não há exemplos pronto para uso no *DORIF* para os mesmos.

4.1.3 Servidor *DORIF*

O servidor construído pelo *DORIF* é simples. Criado com base no *Thrift*, possui somente um método *run()* da interface *Runnable* para inicializar o serviço. Este método é utilizado para que o servidor possa ser inicializado em uma *thread* diferente da principal, viabilizando programas que realizam outras operações além da de servidor do *DORIF*. Inclusive, viabilizando aplicações onde o cliente e o servidor são o mesmo programa.

Para criar um servidor *DORIF*, basta incluir o arquivo *.jar* do *framework* no projeto, criar um objeto do tipo *DORIFServerBuilder*, que implementa o padrão de projetos *Builder* (GAMMA et al., 1994), e utilizar os métodos fornecidos pelo objeto para definir as configurações do servidor e repassar o *DORIFHandler* que será utilizado. No estágio atual, somente estão disponíveis configurações de porta ao qual o servidor será vinculado no sistema e qual protocolo de comunicação do *Thrift* será utilizado.

A princípio, somente um dos protocolos está implementado, o binário compacto simples. Após a configuração do objeto o método *build()* deverá ser chamado para criar uma instância de servidor. Finalmente, uma *Thread* deverá ser criada e a instância do servidor passada como argumento. Neste momento, quando o usuário utilizar o método *start()* da *Thread*, o servidor será iniciado.

4.2 Estudos de Caso e Aplicação

Com base no modelo vetorial nativo do *DORIF*, explanado anteriormente, foi construído um estudo de caso que explora na prática a utilização do sistema. Para criar um exemplo real que explorasse completamente as funcionalidades do sistema, o estudo de caso contempla um servidor e um cliente. Ambos serão descritos nas seções a seguir.

4.2.1 Servidor: Coleção de Texto Simples

O servidor criado para o estudo de caso, desenvolvido em Java, restringe-se a um serviço de indexação e recuperação de arquivos de texto simples no formato (*.txt*). Como fora utilizado o modelo vetorial pronto, nativo do *framework*, somente foram necessárias as implementações de um coletor de documentos, um pré processador e uma classe principal.

A classe principal é responsável somente pela inicialização do servidor. Esta etapa compreende a instanciação de:

- uma fábrica;
- um coletor de documentos;
- um pré processador;
- uma função de ranqueamento;
- um gerenciador de coleção.

Utilizando os módulos anteriores como parâmetros de construção, um *DORIFHandler* que recebe o gerenciador de coleção e um *DORIFServer*, construído com base no *DORIFHandler* criado. Após todo o preparo dos objetos, o método *start()* do servidor é invocado em uma *thread* separada e o programa principal não possui mais instruções para executar.

O pré processador desenvolvido para este servidor possui uma implementação minimalista. De acordo com a interface *IPreProcessor*, deve ser possível requisitar o método de pré processamento utilizando um documento ou uma *String* como argumento. No caso do documento, o método recupera o conteúdo pelo método *getContent()* previsto na interface *IDocument* e realiza uma chamada para si mesmo, porém, utilizando a *String* do conteúdo como argumento. É destacável a utilização da sobrecarga de métodos pela interface *IPreProcessor*. O pré processamento utilizando *String* realiza um tratamento inicial de conversão das letras para minúsculas. Posteriormente é aplicada uma série de substituições de caracteres especiais como os acentuados.

Finalmente, todos os grupos de dois caracteres de espaço ou mais são transformados em um único espaço. Isto é feito com o objetivo de facilitar a quebra em termos. Neste caso, os termos são considerados palavras únicas. Portanto, a quebra é realizada por espaços. O resultado é um *array* de *Strings*. Porém, somente são mantidas aquelas que possuírem dois ou mais caracteres. Isso evita que alguns artigos sejam indexados. Não foram utilizados três caracteres como mínimo, o que retiraria artigos como "os" e "as" pois os documentos de teste estavam em inglês e existem muitas palavras significativas com dois caracteres nesta língua, como por exemplo: *to*, *be*, *do*, *is*, entre outras.

O coletor de documentos por sua vez, implementa a interface *ICollector* do *DORIF*. Como as coleções podem ser dinâmicas, ou seja, documentos podem ser adicionados a qualquer momento, a interface de coletores já prevê a necessidade de funcionamento paralelo do coletor. Isto traduz-se na utilização do padrão *Observer* explicado anteriormente para, assim que novos documentos forem detectados, notificar o gerenciador de coleção e realizar o repasse dos mesmos. Além disso, o estudo de caso desenvolveu um comportamento característico de *thread* durante a escrita do método *run()*, este originado da interface *Runnable* estendida pela interface *ICollector*.

Ao ser iniciado com o método *startCollecting()*, o método *run()* inicia um *loop* teoricamente infinito. Na realidade este *loop* executará enquanto a variável *collect*, booleana,

for verdadeira. Esta variável é definida como falsa apenas quando executa-se o método *stopCollecting()*. Enquanto é executado, o método *run()* verifica um diretório fixo quanto a existência de novos documentos. Para isso, é utilizado um *array* que guarda os nomes de todos os documentos visitados. Embora não seja a implementação mais eficiente para a verificação, é válida para o estudo de caso dado o tamanho da coleção utilizada, quatro documentos a saber.

Após a verificação quanto a novos documentos, caso haja realmente documentos ainda não indexado, a rotina irá lê-los e criar *IDocuments* para armazená-los na coleção. Todos os documentos encontrados são adicionados a um *buffer* de documentos. Este buffer é utilizado como argumento na notificação do gerenciador de coleções pela funcionalidade provida pelo *Observer*. Terminada esta etapa, a *thread* do coletor executa a instrução *Thread.sleep(5000)* o que a faz entrar em esperar por cinco segundos. Uma vez encerrado o período, a verificação do *loop* inicial é realizada novamente e, como explicado anteriormente, caso o método *stopCollecting()* não tenha sido chamado e, portanto, a variável *collect* não esteja falsa, será executado outro ciclo de verificação.

Com todas as classes prontas, é possível compilar e executar várias instâncias do servidor, seja em máquinas diferentes, sejam na mesma máquina. Como é sugerido no artigo do *Chord*, pode ser interessante executar algumas instâncias do serviço na mesma máquina para melhorar a distribuição de chaves e, portanto, a distribuição da carga entre os nós na rede. A imagem 9 ilustra a execução de vários servidores em uma mesma máquina.

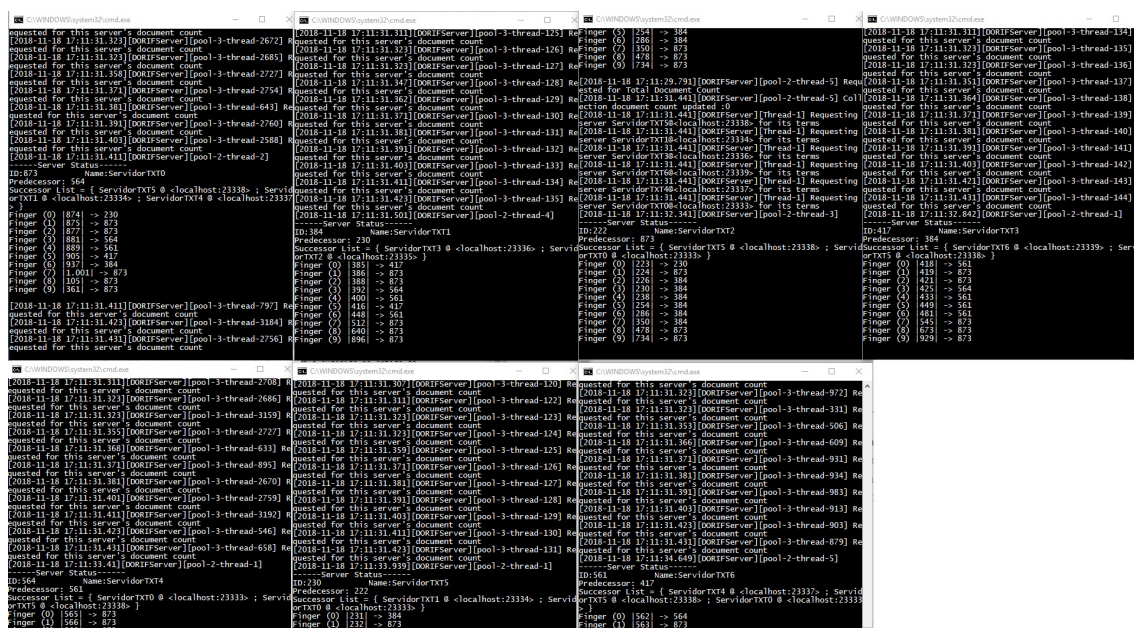


Figura 9 – Sete instâncias de do servidor de estudo de caso operando em uma mesma máquina.

4.2.2 Cliente: Console

O cliente, utilizado como estudo de caso, foi implementado em Python versão 3. A escolha da linguagem possui o propósito de ilustrar, na prática, a possibilidade de criação de clientes em outras linguagens que não a do servidor. Mesmo sendo utilizado um protocolo binário compactado, o *Thrift* se encarrega do trabalho de serializar os dados em uma das pontas e realizar o processo inverso na outra. Durante estes processos, todas as conversões necessárias são realizadas. Conversões estas, que muitas vezes variam com a arquitetura do processador da máquina utilizada, além é claro da linguagem de implementação do programa que utiliza o *Thrift*.

Para a criação do projeto de um cliente, de maneira genérica, basta incluir os arquivos compilados do *Thrift* para a linguagem escolhida. Nestes arquivos, derivados do que foi definido via IDL, estão os métodos que o cliente utilizará para realizar a comunicação com qualquer servidor. O primeiro passo consiste em realizar a conexão com o servidor, o que é cumprido criando-se um *Socket* do *Thrift* e informando os dados de IP e porta de algum dos servidores do anel.

Uma vez conectado a algum dos nós da rede, é possível realizar qualquer solicitação ao servidor. Para tal, no entanto, é preciso, anteriormente, realizar a implementação do comportamento dos métodos que serão chamados. Não há, porém, necessidade de escrever todos os métodos previstos na IDL. Cada programa cliente pode utilizar somente os métodos que forem interessantes para o mesmo. Isto é útil, pois existem métodos que são de controle remoto dos servidores e que estão no mesmo serviço *Thrift*.

O cliente desenvolvido em Python para o estudo de caso, foi baseado no padrão de programas para console, ou terminal. Inicialmente é necessário informar manualmente o IP e porta do servidor para conexão. Esta funcionalidade claramente haveria de ser alterada no caso de uma aplicação para usuários convencionais, porém, foi aplicada para facilitar testes. Uma vez realizada a conexão, é possível submeter requisições ao anel de servidores via comandos textuais.

A base escolhida para teste e demonstração do sistema foi a mesma utilizada no livro "Recuperação de Informação : Conceitos e Tecnologia das Máquinas de Busca" de [Baeza-Yates e Ribeiro-Neto \(2013\)](#). A mesma é exposta na Figura 10

Esta base foi escolhida pois o livro oferece um conjunto de tabelas que possuem todos os valores de frequências e pesos, portanto, servindo como comparativo para validação da funcionalidade do sistema. No entanto reconhece-se que a o conjunto de arquivos é pequeno e seu conteúdo também. Para melhor verificação do correto funcionamento do sistema, seriam requeridos mais testes com maiores conjuntos de documentos.

Para demonstrar a utilização do cliente e do sistema como um todo, novamente sete servidores foram iniciados em uma máquina. Posteriormente, quatro servidores receberam

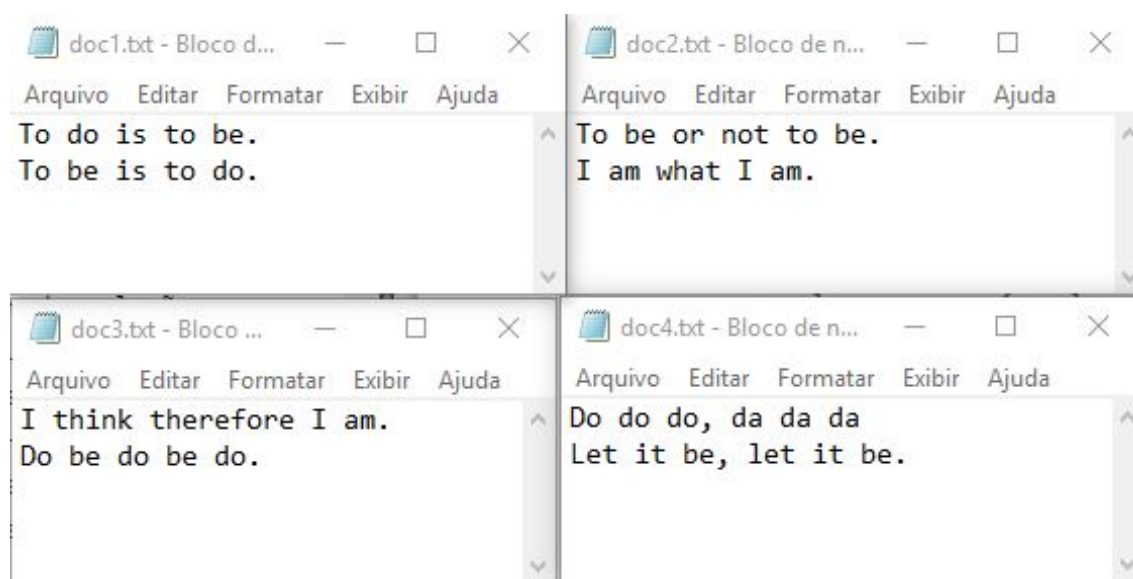


Figura 10 – Base de dados extraída de (BAEZA-YATES; RIBEIRO-NETO, 2013)

um documento cada. A seguir são apresentados alguns dos comandos disponíveis que listam informações à respeito de documentos, termos e seus pesos.

Primeiramente, realizou-se uma conexão, exposta na Figura 11, com um dos servidores que não recebeu nenhum documento. Isto é feito com o propósito de se demonstrar que os dados realmente estão vindo de outros servidores no anel.

```
Digite o IP: localhost
Digite a porta: 23337
Conectado com o Sucesso a localhost : 23337
```

Figura 11 – Conexão com servidor pelo cliente.

Utilizando o comando *GlobalTerms* o cliente solicita ao servidor conectado a listagem de todos os termos globais do sistema, juntamente com suas respectivas informações de ID, frequência e peso IDF. O retorno da requisição pode ser visualizado na imagem 12.

Comparando-se com a tabela de pesos fornecida no livro e exposta na Figura 13, percebe-se que os números condizem. Ao comparar ambas tabelas, nota-se também a ausência do termo *I* na listagem do sistema. Isto ocorreu devido ao pré processamento, citado anteriormente, aceitar somente palavras com dois ou mais caracteres. Este tratamento não fora realizado no processamento utilizado no livro, portanto, houve a discrepância.

```
@-GlobalTerms
Recuperando informações sobre todos os termos de vocabulário geral
```

ID	TERMO	FREQUENCIA	IDF
698	therefore	1	2.000
686	what	1	2.000
855	da	1	2.000
615	do	3	0.415
148	think	1	2.000
210	or	1	2.000
911	to	2	1.000
257	be	4	0.000
251	is	1	2.000
273	it	1	2.000
450	let	1	2.000
534	not	1	2.000
458	am	2	1.000

```
|a-
```

Figura 12 – Resultado da execução do comando *GlobalTerms* no cliente.

#	termo	n_i	$IDF_i = \log(N/n_i)$
1	to	2	1
2	do	3	0,415
3	is	1	2
4	be	4	0
5	or	1	2
6	not	1	2
7	I	2	1
8	am	2	1
9	what	1	2
10	think	1	2
11	therefore	1	2
12	da	1	2
13	let	1	2
14	it	1	2

Figura 13 – Tabela de termos e valores de peso IDF, extraída de (BAEZA-YATES; RIBEIRO-NETO, 2013).

A seguir o comando *DocTerms* foi executado no cliente. Isto solicita ao servidor a listagem de todos os termos por documento, aliados de seus respectivos caminhos para o documento, ID, frequência, peso TF e peso TF-IDF. O resultado é exemplificado pela Figura 14.

```
@-DocTerms
Recuperando informações sobre todos os termos de vocabulário dos documentos
```

DOCUMENTO	ID	TERMO	FREQUENCIA	TF	TFIDF
.\base_livro\doc1.txt	257	be	2	2.000	0.000
.\base_livro\doc1.txt	251	is	2	2.000	4.000
.\base_livro\doc1.txt	911	to	4	3.000	3.000
.\base_livro\doc1.txt	615	do	2	2.000	0.830
.\base_livro\doc3.txt	148	think	1	1.000	2.000
.\base_livro\doc3.txt	257	be	2	2.000	0.000
.\base_livro\doc3.txt	698	therefore	1	1.000	2.000
.\base_livro\doc3.txt	615	do	3	2.585	1.073
.\base_livro\doc3.txt	458	am	1	1.000	1.000
.\base_livro\doc2.txt	534	not	1	1.000	2.000
.\base_livro\doc2.txt	257	be	2	2.000	0.000
.\base_livro\doc2.txt	210	or	1	1.000	2.000
.\base_livro\doc2.txt	686	what	1	1.000	2.000
.\base_livro\doc2.txt	911	to	2	2.000	2.000
.\base_livro\doc2.txt	458	am	2	2.000	2.000
.\base_livro\doc4.txt	257	be	2	2.000	0.000
.\base_livro\doc4.txt	450	let	2	2.000	4.000
.\base_livro\doc4.txt	615	do	3	2.585	1.073
.\base_livro\doc4.txt	273	it	2	2.000	4.000
.\base_livro\doc4.txt	855	da	3	2.585	5.170

Figura 14 – Resultado da execução do comando *DocTerms* no cliente.

Em sequência, para comparação as tabelas correspondentes do livro são expostas nas figuras 15 e 16.

#	termo	$f_{i,1}$	$f_{i,2}$	$f_{i,3}$	$f_{i,4}$	$TF_{i,1}$	$TF_{i,2}$	$TF_{i,3}$	$TF_{i,4}$
1	to	4	2	–	–	3	2	–	–
2	do	2	–	3	3	2	–	2,585	2,585
3	is	2	–	–	–	2	–	–	–
4	be	2	2	2	2	2	2	2	2
5	or	–	1	–	–	–	1	–	–
6	not	–	1	–	–	–	1	–	–
7	I	–	2	2	–	–	2	2	–
8	am	–	2	1	–	–	2	1	–
9	what	–	1	–	–	–	1	–	–
10	think	–	–	1	–	–	–	1	–
11	therefore	–	–	1	–	–	–	1	–
12	da	–	–	–	3	–	–	–	2,585
13	let	–	–	–	2	–	–	–	2
14	it	–	–	–	2	–	–	–	2
Tamanho do documento (# palavras)		10	11	10	12				

Figura 15 – Tabela de termos, frequência e valores de TF, retirada de (BAEZA-YATES; RIBEIRO-NETO, 2013).

Encerrando os comandos de listagem de informações da coleção, o comando *DocInfo* foi submetido para obter a relação entre documentos e servidores e os coeficientes de normalização dos respectivos documentos. O resultado é demonstrado na Figura 17.

#	termo	d_1	d_2	d_3	d_4
1	to	3	2	–	–
2	do	0,830	–	1,073	1,073
3	is	4	–	–	–
4	be	–	–	–	–
5	or	–	2	–	–
6	not	–	2	–	–
7	I	–	2	2	–
8	am	–	2	1	–
9	what	–	2	–	–
10	think	–	–	2	–
11	therefore	–	–	2	–
12	da	–	–	–	5,170
13	let	–	–	–	4
14	it	–	–	–	4
Tamanho do documento (normas dos vetores)		5,068	4,899	3,762	7,738

Figura 16 – Tabela de valores de TF-IDF e norma, extraída de (BAEZA-YATES; RIBEIRO-NETO, 2013).

```
@-DocsInfo
Recuperando informações de todos os documentos....
```

SERVIDOR	ID	NOME	CAMINHO	NORMA
ServidorTXT0(873)	0	.\base_livro\doc1.txt	.\base_livro\doc1.txt	5.068
ServidorTXT2(222)	0	.\base_livro\doc3.txt	.\base_livro\doc3.txt	3.186
ServidorTXT1(384)	0	.\base_livro\doc2.txt	.\base_livro\doc2.txt	4.472
ServidorTXT3(417)	0	.\base_livro\doc4.txt	.\base_livro\doc4.txt	7.738

```
@-
```

Figura 17 – Resultado da execução do comando *DocsInfo* no cliente.

Observa-se a variação no valor da norma nos documentos 2 e 3 como consequência da não consideração do termo *I* nos vocabulários do sistema. Na imagem, também é possível comprovar que os quatro documentos que compõem a coleção foram realmente espalhados em quatro servidores distintos. Todos recebem o ID 0 por serem os únicos documentos em cada nó.

Uma vez que os documentos foram indexados no sistema, é possível realizar consultas à coleção. No cliente, isto pode ser realizado pelo comando *query* seguido da consulta, como pode ser visualizado na Figura 18.

```
@-query to do
Consultando servidor para to do...
```

SCORE	NOME	ID	DESCRICAO	SERVER ID	SERVER NAME
0.609	.\base_livro\doc1.txt	0		873	ServidorTXT0
0.413	.\base_livro\doc2.txt	0		384	ServidorTXT1
0.129	.\base_livro\doc3.txt	0		222	ServidorTXT2
0.053	.\base_livro\doc4.txt	0		417	ServidorTXT3

```
@-
```

Figura 18 – Resultado da consulta *to do*.

Como resultado, obtém-se o ranqueamento dos documentos em ordem decrescente de *score*, ou seja, do que o sistema julgou mais relevante para o menos relevante. Se for realizada uma comparação entre a consulta, *to do*, com o documento *doc1.txt*, disponível na Figura 10, percebe-se o porquê deste ter sido classificado como o primeiro. Dentre todos os documentos é o que melhor se assemelha à necessidade de informação submetida.

5 Conclusão

O objetivo principal deste projeto consistia em construir um *framework* para desenvolvimento de aplicações com RI distribuída. Este *framework*, nomeado *DORIF*, foi então implementado de forma portátil, podendo ser utilizado em vários sistemas operacionais, desde que haja suporte à linguagem Java, no caso de um servidor. No caso dos clientes da aplicação, existe também amplo suporte a sistemas operacionais, mas principalmente às linguagens de programação em que podem ser desenvolvidos. Utilizando um protocolo de comunicação binário as mensagens entre clientes e servidores e dos servidores entre si tornam-se menores e mais eficientes. As facilidades de comunicação foram decorrentes da utilização do *Thrift*, *framework* da Apache.

A distribuição do sistema e seu gerenciamento foram possibilitados pela implementação do protocolo *Chord* desenvolvido no Massachusetts Institute of Technology (MIT). Sua utilização proveu a criação de uma rede *peer-to-peer* de servidores que dividem a carga de um vocabulário global e tratam necessidades de informação dos usuários de forma distribuída e independente, de acordo com suas bases de documentos individuais.

Dentro dos servidores, a camada lógica do *DORIF* fornece a possibilidade de implementação de um sistema de RI personalizável, modularizado e com uma implementação pronta do modelo vetorial, comumente usado. Sua simplicidade permite também a aplicação no ensino. Com este *framework*, é possível implementar vários módulos utilizando diferentes teorias e métodos da área de organização e recuperação da informação. Torna-se assim, uma notável ferramenta para contribuir com o ensino em sala de aula, podendo ser utilizada tanto por professores quanto por alunos.

A facilidade de uso e implementação das interfaces, a simplicidade de configuração e utilização dos servidores e a capacidade de escalabilidade, são características que reforçam a aplicação do *DORIF* em ambientes comerciais e educacionais. Isto se remete também ao cumprimento dos objetivos de flexibilidade e descomplicação do desenvolvimento de sistemas de RI distribuídos. Desta forma, a maior preocupação do usuário durante a utilização do *framework* recai sobre suas próprias regras de negócio.

No entanto, existem vários pontos que não puderam ser atingidos neste trabalho. Quanto à distribuição da informação, e como sistema distribuído de maneira geral, faltam funcionalidades essenciais para o bom funcionamento e manutenção. Um exemplo é que não há replicação de dados, portanto, quando um nó é desligado da rede, com ou sem aviso prévio, os dados que estavam contidos no mesmo se perdem até que este seja religado. Com replicação, não haveria interrupção de serviço para estes dados, uma vez que um ou mais servidores poderiam responder pelo que se encontra *offline*.

Na parte da RI, ainda não há implementação para as etapas opcionais de expansão e realimentação da consulta. Estas etapas, embora não sejam obrigatórias, estão presentes em sistemas grandes atuais como o Google, já citado anteriormente. Além disso, deve ser criada um tipo de módulo gancho que possa ser acoplado no sistema para realizar tarefas pós ranqueamento, como é o caso do *PageRank* da Google. Este passo adicional é o responsável por reclassificar os sites recuperados de acordo com indicadores de popularidade dos mesmos.

Desta forma, trabalhos futuros incluem o preenchimento dessas lacunas e a adição de outras funcionalidades além de uma revisão completa no desempenho dos sistemas. A preocupação com o desempenho é uma parte fundamental e complexa, porém, que deve ser muito explorada devido ao fato de tratar-se de um sistema distribuído. Embora possa ser facilmente escalável com o *Chord*, mais estudos são necessários para avaliar o comportamento de resposta do sistema sob grandes demandas.

Espera-se também que o *DORIF* possa ser explorado como ferramenta educacional com o intuito de facilitar e aprimorar o ensino das disciplinas de Organização e Recuperação da Informação e Sistemas Distribuídos. Disciplinas estas presentes nos cursos de Sistemas de Informação, Ciência da Computação e afins e que possuem um nível de complexidade maior, dificultando o ensino pelos professores e a aprendizagem pelos alunos.

Referências

- Apache Lucene. *Apache Lucene*. 2017. <<https://lucene.apache.org/>>. <<https://lucene.apache.org/>>, Acesso em: 11/05/2018. Citado 4 vezes nas páginas 6, 16, 19 e 28.
- BAEZA-YATES, R.; RIBEIRO-NETO, B. *Recuperação de Informação-: Conceitos e Tecnologia das Máquinas de Busca*. [S.l.]: Bookman Editora, 2013. Citado 8 vezes nas páginas 8, 9, 10, 41, 42, 43, 44 e 45.
- BIAŁECKI, A.; MURI, R.; INGERSOLL, G. Apache Lucene 4. *Proceedings of the SIGIR 2012 Workshop on Open Source Information Retrieval*, p. 17–24, 2012. ISSN 0163-5840. Citado na página 16.
- DEWEY, M. *Decimal Classification and Relative Index for Libraries: Clippings, Notes, Etc.* [S.l.]: Library bureau, 1899. Citado na página 8.
- DORIF GitHub Repository. 2018. <<https://github.com/ArthurDeveloper/dorif>>. Acesso em: 03/01/2018. Citado na página 21.
- ECKERT, J. J. P.; MAUCHLY, J. W. *Electronic Numerical Integrator and Computer*. [S.l.]: Google Patents, 1964. Citado na página 13.
- EDWARD, S. H. *Means for Compiling Tabular and Statistical Data*. [S.l.]: Google Patents, 1920. Citado na página 8.
- ENSER, P. The Evolution of Visual Information Retrieval. *Journal of Information Science*, v. 34, n. 4, p. 531–546, 2008. ISSN 01655515. Citado na página 10.
- EVERNOTE. 2018. <<https://evernote.com/intl/pt-br>>. Acesso em: 03/01/2018. Citado na página 28.
- FAYAD, M.; SCHMIDT, D. C. Object-Oriented Application Frameworks. *Communications of the ACM*, v. 40, n. 10, p. 32–38, 1997. ISSN 00010782. Citado na página 15.
- FOX, M. S. An Organizational View of Distributed Systems. *IEEE Transactions on Systems, Man and Cybernetics*, v. 11, n. 1, p. 70–80, 1981. ISSN 21682909. Citado na página 14.
- GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. [S.l.]: Pearson Education, 1994. (Addison-Wesley Professional Computing Series). ISBN 9780321700698. Citado 3 vezes nas páginas 22, 24 e 38.
- Google. *Google*. 2018. <<http://google.com>>. Acesso em: 11/05/2018. Citado 2 vezes nas páginas 6 e 14.
- iText. 2018. <<https://itextpdf.com/>>. Acesso em: 03/01/2018. Citado na página 15.
- JONES, K. S. A Statistical Interpretation of Term Specificity and its Retrieval. *Journal of Documentation*, v. 28, n. 1, p. 11–21, 1972. ISSN 0022-0418. Citado na página 12.

- JQueryUI. 2018. <<https://jqueryui.com/>>. Acesso em: 03/01/2018. Citado na página 15.
- KO, Y. A Study of Term Weighting Schemes Using Class Information for Text Classification. *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval - SIGIR '12*, p. 1029, 2012. Citado na página 13.
- LAN, M. et al. A Comprehensive Comparative Study on Term Weighting Schemes for Text Categorization with Support Vector Machines. *Special interest tracks and posters of the 14th international conference on World Wide Web - WWW '05*, p. 1032, 2005. Citado na página 13.
- LUHN, H. P. A Statistical Approach to Mechanized Encoding and Searching of Literary Information. *IBM Journal of Research and Development*, v. 1, n. 4, p. 309–317, 1957. ISSN 0018-8646. Citado na página 12.
- NANUS, B. The Use of Electronic Computers for Information Retrieval. *Bulletin of the Medical Library Association*, Medical Library Association, v. 48, n. 3, p. 278, 1960. Citado na página 8.
- OAuth. 2012. <<https://tools.ietf.org/html/rfc6749>>. Acesso em: 03/01/2018. Citado na página 15.
- OSIŃSKI, S.; WEISS, D. Carrot 2 : Design of a Flexible and Efficient Web Information Retrieval Framework. *Proceedings of the Third international conference on Advances in Web Intelligence*, p. 439–444, 2005. ISSN 03029743. Citado na página 17.
- PINTEREST. 2018. <<https://br.pinterest.com/>>. Acesso em: 03/01/2018. Citado na página 28.
- POWERED By Thrift. 2018. <<https://thrift.apache.org/about#powered-by-apache-thrift>>. Acesso em: 03/01/2018. Citado na página 28.
- RAMOS, J. *Using TF-IDF to Determine Word Relevance in Document Queries*. 2003. Citado na página 13.
- RAO, R. et al. The Information Grid: A Framework for Information Retrieval and Retrieval-Centered Applications. *ACM*, p. 23–32, 1992. Citado na página 18.
- ROBERTSON, S.; ZARAGOZA, H. *The Probabilistic Relevance Framework: BM25 and Beyond*. 2009. 333–389 p. Citado na página 16.
- ROOCK, S.; WOLF, H.; ZÜLLIGHOVEN, H. Frameworking. In: *IRIS*. [S.l.: s.n.], 1998. v. 21, p. 743–758. Citado na página 15.
- SALTON, G.; YANG, C. On The Specification of Term Values in Automatic Indexing. *Journal of Documentation*, v. 29, n. 4, p. 351–372, 1973. Citado na página 12.
- SANDERSON, M.; CROFT, W. B. The History of Information Retrieval Research. *Proceedings of the IEEE*, v. 100, p. 1444–1451, 2012. ISSN 0018-9219. Citado na página 6.

Solr Features. *Solr Features*. 2017. <<http://lucene.apache.org/solr/features.html>>. Acesso em: 23/06/2018. Citado 3 vezes nas páginas 17, 19 e 28.

STOICA, I. et al. Chord : A Scalable Peer-to-peer Lookup Service for Internet Applications. *SIGCOMM*, 2001. Citado 3 vezes nas páginas 29, 30 e 32.

TAUBE, M.; GULL, C. D.; WACHTEL, I. S. Unit Terms in Coordinate Indexing. *Journal of the Association for Information Science and Technology*, Wiley Online Library, v. 3, n. 4, p. 213–218, 1952. Citado na página 8.

Thrift. *Apache Thrift*. 2017. <<https://thrift.apache.org/>>. Acesso em: 23/06/2018. Citado 2 vezes nas páginas 7 e 28.

UBER. 2018. <<https://www.uber.com/br/pt-br/>>. Acesso em: 03/01/2018. Citado na página 28.

WESTERINEN, A.; BUMPUS, W. The Continuing Evolution of Distributed Systems Management. *Management*, n. 11, p. 2256–2261, 2003. Citado na página 13.

Yahoo Vespa Docs. *Features: What is Vespa?* 2017. <<https://docs.vespa.ai/documentation/features.html>>. Acesso em: 23/06/2018. Citado 2 vezes nas páginas 17 e 19.

ZACHMAN, J. A. A Framework for Information Systems Architecture. *IBM Systems Journal*, v. 26, n. 3, p. 454–470, 1987. ISSN 0018-8670. Citado na página 15.

Zookeeper. 2008. <<https://zookeeper.apache.org/>>. Acesso em: 23/06/2018. Citado na página 17.