UNIVERSIDADE DO VALE DO ITAJAÍ
CENTRO DE CIÊNCIAS TECNOLÓGICAS DA TERRA E DO MAR
CURSO DE CIÊNCIA DA COMPUTAÇÃO

UMA ARQUITETURA FLEXÍVEL PARA A COLETA DE DADOS

Área de Recuperação de Informação

por

Lucas Nazário dos Santos

Eros Comunello, Dr. rer. nat.
Orientador

Alexandre Leopoldo Gonçalves, Dr.
Co-orientador

São José (SC), dezembro de 2009

UNIVERSIDADE DO VALE DO ITAJAÍ
CENTRO DE CIÊNCIAS TECNOLÓGICAS DA TERRA E DO MAR
CURSO DE CIÊNCIA DA COMPUTAÇÃO

UMA ARQUITETURA FLEXÍVEL PARA A COLETA DE DADOS

Área de Recuperação de Informação

por

Lucas Nazário dos Santos

Relatório apresentado à Banca
Examinadora do Trabalho de Conclusão do
Curso de Ciência da Computação para
análise e aprovação.
Orientador: Eros Comunello, Dr. rer. nat.

São José (SC), dezembro de 2009

DEDICATÓRIA

Ao meu pai.

AGRADECIMENTOS

Aos meus pais e a minha avó, os pilares que condicionaram a materialização dos meus maiores e mais importantes desejos.

Ao meu grande amigo Alfredo Branco pelo braço estendido e mão aberta nos momentos mais inóspitos da minha vida.

Ao professor Eros Comunello pela orientação cirúrgica que culminou não só no presente trabalho, mas em uma relação interpessoal especial e única.

Ao professor Alexandre Gonçalves pela exposição do tema de pesquisa e co-orientação.

Ao professor Vinícius Kern pelas oportunidades de pesquisa no curso da graduação, por minha iniciação na vida acadêmica e pelo exemplo de pessoa integra e ética que me faz querer ser alguém melhor a cada dia.

Ao Instituto Stela, à Univali e a todos os colegas integrantes dessas duas instituições que ofertaram um ambiente mais do que propício ao desenvolvimento desta pesquisa.

Aos professores Rafael Cancian, Adhemar Filho, Edson Bez e Anita Fernandes, e também ao acadêmico Helton Krauss pelos valiosos apontamentos realizados durante TCC e que foram essenciais à maturação e conclusão do trabalho.

Ao Fabiano Beppler que, além de ser o maior fator recente de mudança, crescimento e inspiração na minha vida, me fez sentir, por incontáveis vezes, o prazer de ter um irmão mais velho.

SUMÁRIO

LISTA DE ABREVIATURAS

ATOM	Atom Syndication Format
API	Application Programming Interface
CPU	Central Processing Unit
CVS	Concurrent Version System
DNS	Domain Name System
FTP	File Transfer Protocol
HD	Hard Drive
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol over Secure Socket Layer
ICMP	Internet Control Message Protocol
IDE	Integrated Development Environment
IP	Internet Protocol
JCIFS	Java Common Internet File System
LAN	Local Area Network
MD5	Message-Digest Algorithm 5
MIME	Multipurpose Internet Mail Extensions
MP3	MPEG-1 Audio Layer 3
MPEG	Motion Picture Expert Group
NNTP	Network News Transfer Protocol
NTFS	New Technology File System
PDF	Portable Document Format
POI	Poor Obfuscation Implementation
RAM	Random Access Memory
RPM	Rotations Per Minute
RSS	Really Simple Syndication
SHA	Secure Hash Algorithm
SMB	Server Message Blocking
SVN	Subversion
TCC	Trabalho de Conclusão de Curso
TCP	Transmission Control Protocol
UNIVALI	Universidade do Vale do Itajaí
UML	Unified Modeling Language
URL	Uniform Resource Location
WWW	World Wide Web
WAIS	Wide Area Information Server

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE EQUAÇÕES

RESUMO

SANTOS, Lucas Nazário dos. **Uma Arquitetura Flexível Para a Coleta de Dados.** São José, 2009. Trabalho de Conclusão de Curso (Graduação em Ciência da Computação)–Centro de Ciências Tecnológicas da Terra e do Mar, Universidade do Vale do Itajaí, São José, 2009.

A coleta de dados é desafiadora pela quantidade de cenários de coleta possíveis. A abundância de cenários é o resultado da combinação dos inúmeros tipos de documentos existentes pelas diversas fontes onde esses podem estar presentes. Nesse contexto, o problema é encontrar e disponibilizar documentos que atendam uma necessidade de coleta específica. A proposta do presente trabalho é a concepção de uma arquitetura para a coleta de dados que permita a recuperação de documentos relevantes a partir de uma necessidade de coleta qualquer. Para atestar a resiliência da arquitetura, técnicas de avaliação foram utilizadas como a execução de cenários de coleta pré-moldados e a checagem das características da solução proposta contra características desejáveis a coletores flexíveis. Os resultados mostram uma arquitetura flexível ao ponto de permitir a inspeção de variadas fontes de dados por distintos tipos de documentos, desse modo comportando uma ampla gama de necessidades de coleta.

Palavras-chave: coleta de dados, coletor de dados, recuperação de informação.

ABSTRACT

The crawling of data is challenging due to the amount of possible crawling scenarios. The multitude of scenarios results from the combination of the great deal of existing documents by the diversity of sources where those documents can be hosted. In this context, the problem is to find and make available documents that address a specific crawling necessity. This work's proposal is the conception of a data crawling architecture that allows the retrieval of relevant documents as consequence of some crawling necessity. To attest the proposal's resilience, assessment techniques were applied like the execution of preconceived crawling scenarios and the appraisal of the solution's features against flexible-crawlers desired functionalities. The results show a flexible architecture that allows the inspection of a variety of sources for distinct types of documents, this way addressing an ample range of crawling necessities.

***Keywords**: crawling, crawler, information retrieval.*

1 INTRODUÇÃO

A coleta de dados permeia várias outras atividades. Estatísticos muitas vezes necessitam coletar e analisar dados para a criação de modelos que apontem um caminho na direção da solução para um determinado problema (ARMITAGE; BERRY; MATTHEWS, 2002). Em outra situação, a prática de disciplina médica é melhorada pelo suporte de sistemas computacionais que coletam, transformam e entregam informação acerca de casos clínicos (BROKEL; HARRISON, 2009). Ainda, engenheiros e cientistas da computação tentam tornar disponíveis grandes massas de dados que precisam antes ser alcançadas por alguma atividade de coleta (BRIN; PAGE, 1998).

Os exemplos acima citados são uma pequeníssima fração da quantidade de processos imbuídos por algum mecanismo de coleta. Tal ortogonalidade confere a essa prática elementos suficientes para que ela seja tratada isoladamente, podendo ser aplicada a várias situações distintas como mostra a Figura 1.

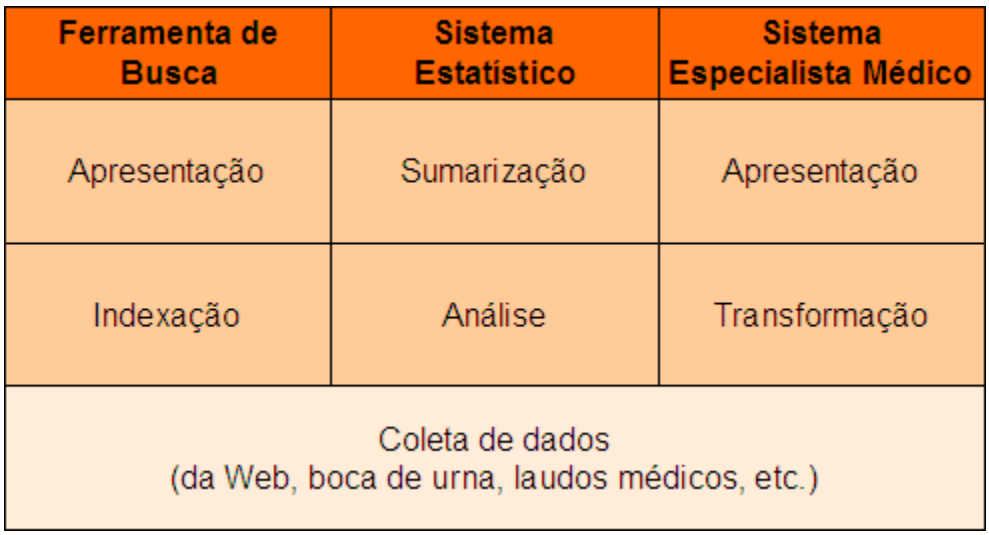


Figura 1. Coleta de dados como parte de outros sistemas.

O processo de coleta de dados é amplo, se estendendo desde a coleta manual como, por exemplo, as famosas “bocas de urna”, até agentes computacionais que navegam pela Web por dados relacionados a um determinado tópico (CHAKRABARTI; BERG; DOM, 1999). Deste modo, para tratar o problema da coleta como atividade isolada dos processos onde atua, é necessário delimitar o seu escopo.

No contexto do presente trabalho, a coleta, também tratada na literatura como *crawling*, é um processo computacional que automaticamente recupera dados alcançados por uma variedade de

protocolos. Conceitos semelhantes são apresentados por Cho, Garcia-Molina e Page (1998) e Gomes e Silva (2008). Os dados são representados por arquivos, e os protocolos delimitam quais fontes de informação podem ser inspecionadas. Por exemplo, caso o protocolo HTTP (*Hypertext Transfer Protocol*) seja suportado, grande parte dos arquivos residentes na Web poderão ser encontrados. Já o suporte ao protocolo SMB (*Server Message Blocking*) possibilitará que muito dos arquivos compartilhados por computadores utilizando sistemas operacionais como Linux e Windows sejam coletados.

A Figura 2 mostra um exemplo de coleta automática de imagens situadas em diferentes fontes de informação. Primeiramente, o coletor, também chamado de *crawler*, *spider*, *ant*, *worm* e *robot*, é configurado com um conjunto de sementes que indicam as fontes de dados **1**. Além das sementes, também é indicado ao coletor o tipo de dado de interesse, nesse caso imagens **2**. Após configurar o coletor com essas duas informações, ele automaticamente inspeciona fontes de informação como a Web, o disco da máquina local, repositórios FTP (*File Transfer Protocol*), sistemas de controle de versão, além de arquivos compartilhados por outros computadores residentes na mesma LAN (*Local Area Network*) **3**, integrando as imagens que encontra em um repositório único para serem posteriormente processadas **4**.

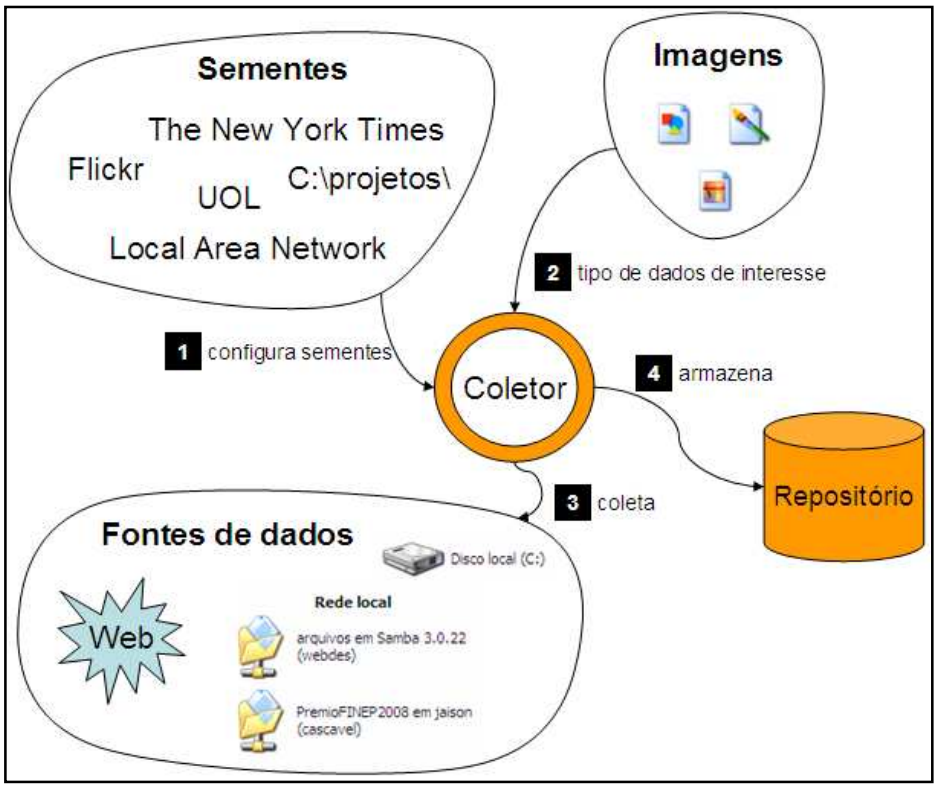


Figura 2. Exemplo de coleta automática de imagens.

A coleta de imagens apresenta indícios da riqueza de cenários com os quais um coletor pode se deparar. Além desse tipo de documento, uma infinidade de outros são encontrados, podendo esses ser armazenados em variadas fontes de informação, cada qual apresentando desafios de coleta únicos. Quando esses dois aspectos (diferentes tipos de documento e distintas fontes de informação) são combinados, a quantidade de cenários gerados é significativa, e um coletor extensível precisa estar habilitado a cooperar com grande parte deles.

É a partir desse contexto que foi proposta a modelagem de uma arquitetura de coleta extensível e modular, com cada módulo podendo ser configurado ou substituído para que um conjunto significativo de necessidades de coleta possa ser referenciado. A arquitetura foi implementada e levou em consideração a fácil integração com soluções existentes, ou seja, é possível operacionalizar o coletor como componente de um sistema maior.

Cenários de coleta foram descritos e cada um deles foi abordado utilizando o coletor construído neste trabalho. Ainda, características descritas pela literatura necessárias a coletores flexíveis foram utilizadas na avaliação da arquitetura.

Espera-se que o coletor edificado sobre a arquitetura proposta seja flexível o suficiente para endereçar distintas necessidades de coleta e que possa ser facilmente operacionalizado como parte de outros sistemas.

1.1 PROBLEMATIZAÇÃO

1.1.1 Formulação do Problema

A coleta de dados é parte integrante de muitos sistemas, essencial para a operacionalização desses. Exemplos são buscadores como o da Google e Yahoo!, softwares agregadores de notícias e sistemas de *clipping*, todos atuando sobre dados coletados de variadas fontes de informação.

Esses exemplos contribuem para a criação de cenários de coleta únicos, produtos da combinação entre a grande variedade de tipos de documento que podem ser coletados pela abundância de fontes provedoras de dados. O grande número de cenários pode ser exemplificado através de casos como: a coleta de artigos científicos (tipo de documento) a partir de bases de

periódicos (fonte de dados), caso do Google Scholar[1]; a recuperação de notícias (tipo) oriundas de *sites* de jornais (fonte), como acontece com o Yahoo! News[2] e Google News[3]; ou mesmo a coleta da maioria dos documentos (tipo) existentes na Web (fonte), caso de buscadores como o da Google[4], e Microsoft[5]. As combinações e os cenários que se desenrolam delas são virtualmente infinitos.

Ainda, o problema da coleta é amplificado por necessidades específicas desse tipo de atividade. Por exemplo, caso seja necessário coletar grandes massas de documentos contidos da Web, é preciso organizar computadores para que trabalhem simultaneamente, recuperando um grande número de documentos em um pequeno espaço de tempo. Porém, caso existam outros usuários dependentes da rede onde o coletor atua, a extenuação deste recurso pelo coletor pode configurar uma má prática, tendo o processo de coleta que se adaptar, provavelmente atuando mais intensamente em determinados períodos como pela madrugada.

Outros exemplos de necessidades que podem se manifestar são: (i) a coleta focada por documentos relevantes a um determinado tópico; (ii) a coleta eficiente, onde documentos importantes são recuperados primeiramente durante o processo; (iii) a coleta distribuída, que busca a integração de diversos computadores para a realização de um objetivo comum; (iv) a polidez, onde o coletor é cauteloso ao contatar servidores de dados com o intuito de não exaurir seus recursos; (v) a sincronização dos dados coletados para que, com o passar do tempo, continuem espelhando o estado das fontes de informação; (vi) a tolerância a falhas, muito desejada por permitir que o coletor continue seu trabalho na presença de algum problema; e (vii) o tratamento aos documentos já recuperados para que, em uma coleta posterior, o coletor possa decidir que atitudes tomar como, por exemplo, atualizá-los ou ignorá-los.

A coleta de variados tipos de dados contidos em distintas fontes de informação, atentando para necessidades específicas de coleta e também para a integração do coletor a outros sistemas, é o problema de pesquisa a ser tratado neste trabalho.

---

[1] http://scholar.google.com/

[2] http://news.yahoo.com/

[3] http://news.google.com/

[4] http://www.google.com/

[5] http://www.bing.com/

1.1.2 Solução Proposta

Um contexto composto por diferentes tipos de documento e fontes de dados, repleto por necessidades específicas de coleta, anseia por um coletor robusto e flexível, suficientemente resiliente tanto para se adaptar a uma ampla gama de cenários de coleta como para se moldar a diversos tipos de sistema, funcionando como parte desses.

Para este fim, foi concebida uma arquitetura genérica e flexível para a coleta de dados acompanhada de uma implementação utilizada na sua avaliação. A arquitetura trata questões como a coleta de dados presente em distintas fontes de informação, a recuperação de variados tipos de documentos contidos em tais fontes, a integração do coletor com outros sistemas, e o entendimento de necessidades específicas de coleta como, por exemplo, a coleta focada, distribuída, e a sincronização dos dados coletados.

1.2 OBJETIVOS

1.2.1 Objetivo Geral

O objetivo geral deste trabalho é conceber e materializar, na forma de um sistema computacional, uma arquitetura flexível para a coleta de dados que possa ser operacionalizada como parte de outros sistemas.

1.2.2 Objetivos Específicos

- Pesquisar e analisar as diferentes soluções para a coleta de dados;
- Compreender as características necessárias à realização de um coletor extensível;
- Modelar conceitualmente a arquitetura;
- Implementar a arquitetura através de um sistema computacional;
- Testar e avaliar a implementação da arquitetura; e
- Documentar o desenvolvimento e os resultados.

1.3 Metodologia

Este trabalho apresenta uma arquitetura genérica para a coleta de diferentes tipos de dados situados em distintas fontes de informação. São exemplos desta afirmação a coleta de grandes

massas de documentos a partir da Web, de arquivos armazenados em uma Intranet ou dados espalhados em máquinas locais. Para que o coletor entenda esses e outros cenários de coleta, sendo capaz de inspecionar boa parte das fontes de informação disponíveis, é imprescindível que ele seja flexível, coletando dados que reflitam a necessidade do usuário. Este fim requereu a conclusão de algumas etapas.

A primeira delas foi a análise de soluções similares, tanto para a aquisição de conhecimento fundamental sobre o domínio do problema quanto para a obtenção de dados acerca da execução das soluções existentes.

Após o entendimento de soluções similares, tanto os requisitos para a modelagem da arquitetura quanto para a expressão dela através de um sistema coletor foram colhidos e explicitados. Eles proveram um caminho seguro, um norte que guiou o desenvolvimento do software.

A materialização dos requisitos através de sistema computacional, o coletor, foi precedida por uma etapa de avaliação tecnológica. Os requisitos, grande parte das vezes, desconsideram a solução tecnológica que será adotada para concretizá-los. Sendo assim, um momento para a análise de tecnologias existentes foi demandado.

Tendo em mãos o entendimento de soluções similares, os requisitos e também os detalhes tecnológicos necessários à implementação do trabalho, foi preciso realizar a análise e projeto da arquitetura e codificá-la em um sistema computacional, o coletor de dados. A avaliação da proposta, com a execução de cenários de coleta pelo coletor, foi efetuada concomitantemente à redação do texto do TCC.

1.4 Estrutura do trabalho

O trabalho está estruturado em quatro capítulos. A introdução (Capítulo 1) apresenta uma visão geral do trabalho. Na fundamentação teórica (Capítulo 2) é feita uma revisão bibliográfica acerca dos conceitos que permearam esta pesquisa. Esta etapa oferta formalizações sobre uma arquitetura genérica para a coleta de dados seguida da exposição de aspectos necessários à sua flexibilização para que ela possa cooperar com uma ampla gama de necessidades de coleta. A fundamentação teórica ainda se desdobra em cenários de aplicação da arquitetura e trabalhos correlatos. O projeto (Capítulo 3) oferece a diagramação do sistema a ser desenvolvido. Nesta

seção, a UML (*Unified Modeling Language*) é utilizada para tal propósito. O desenvolvimento do projeto (Capítulo 4) trata da materialização da arquitetura em um sistema coletor e da avaliação da solução proposta. As conclusões e trabalhos futuros (Capítulo 5) são evidenciadas ao final do trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

A coleta de dados permeia diversas outras atividades. Esse entremear usualmente se dá pela utilização da atividade de coleta como provedora de dados para diversos outros componentes, dados esses que serão, grande parte das vezes, processados e armazenados para posterior utilização. Esse é, grosso modo, o desenrolar de buscadores como o da Google e Yahoo!. Um processo de coleta inicialmente varre a Web por dados de interesse, transformando esses e armazenando-os em estruturas conhecidas por índices que são insumo para algoritmos de recuperação de informação encapsulados através de interfaces inteligíveis por humanos (KORFHAGE, 1997).

A coleta de dados, também conhecida na literatura por *crawling*, no contexto da Web é conceituada como "um programa que automaticamente coleta e armazena páginas Web, frequentemente para buscadores". Entendimentos similares são destacados por Cho, Garcia-Molina e Page (1998) e Kritikopoulos, Sideri e Stroggilos (2004). Porém, apesar do foco da coleta estar sobre a Web, notavelmente pelas características singulares desse meio, que estressam o desenvolvimento de um coletor ao limite pela grande quantidade de dados disponível e também pela falta de padrões, existem outras fontes de dados suficientemente ricas para receberem fração da atenção que é despendida com o desenvolvimento de coletores.

Além da Web, dados estão disponíveis em bancos de dados, em repositórios FTP (*File Transfer Protocol*), compartilhados por computadores de uma mesma LAN (*Local Area Network*) ou entranhados em diretórios dentro de uma máquina local, para citar alguns exemplos. Apesar da quantidade de locais onde dados podem ser armazenados ser significativa, é provável que novos sejam visionados para acomodar tanto a crescente quantidade de informação digital como o anseio dos usuários por espaços de armazenamento que melhor se adaptem às suas necessidades. Hoje, dados digitais estão espalhados por hardwares que vão de máquinas fotográficas a fitas magnéticas, passando por *pen-drivers* e tocadores de MP3 (*MPEG-1 Audio Layer 3*), todos acessíveis através de uma pluralidade de protocolos.

É a partir deste cenário, composto por diversas fontes de dados, cada qual armazenando uma infinidade de tipos de documentos, que uma arquitetura flexível para a coleta de dados deve ser desenvolvida. Ela precisa ser suficientemente resiliente para atender a uma ampla gama de necessidades, ou seja, dada uma pretensão de coleta, um agente

computacional inspeciona as fontes de dados necessárias e retorna ao usuário os dados que esse espera.

Os conceitos fundamentais à concepção dessa arquitetura são descritos nas seções que seguem. Primeiro, uma arquitetura genérica para a coleta automática de dados será retratada. Logo após, aspectos essenciais a uma arquitetura flexível para a coleta de dados serão delineados, com a remodelação da arquitetura genérica para acomodá-los. Além disso, aplicações da coleta de dados serão exploradas através de exemplos encontrados na literatura acadêmica, com a posterior exposição de trabalhos relacionados.

2.1 Arquitetura Genérica Para a Coleta de Dados

A Figura 3, adaptada de Liu (2009, p.275), esboça uma arquitetura genérica para a coleta de dados automática. Os retângulos denotam os elementos arquiteturais, seus componentes, enquanto as setas e os números indicam a sequência com que a etapa é realizada.

Na imagem, o processo de coleta é iniciado pela adição de sementes **1**, que são referências para documentos ou recursos armazenados em alguma fonte. Exemplos são páginas HTML (*Hypertext Markup Language*) na Web ou arquivos PDF (*Portable Document Format*) localizados no disco da máquina local. Cada semente é usualmente representada por uma URL (*Uniform Resource Location*), e essas são dispostas diretamente dentro do *frontier*, o módulo responsável pela manutenção da lista de URLs por serem visitadas **2**. A cada iteração uma URL é removida do *frontier* **3** e o conteúdo do documento apontado por ela é recuperado **4**. Esse conteúdo passa por um processo de filtragem de acordo com as necessidades do usuário **5** que pode definir, por exemplo, a eliminação dos documentos que não contenham determinada palavra-chave. Logo após, duas tarefas são executadas. Primeiro, o conteúdo é inspecionado na busca por novas URLs **6**. Nesse passo, cada nova URL é enviada para um novo processo de filtragem **7** e logo após armazenada dentro do *frontier* **8** caso não seja eliminada. Depois, o coletor notifica quaisquer módulos responsáveis por manipular os documentos coletados **9** e continua iterando sobre o *frontier* até que URLs não mais existam ou que algum outro critério de parada seja atingido **10**.

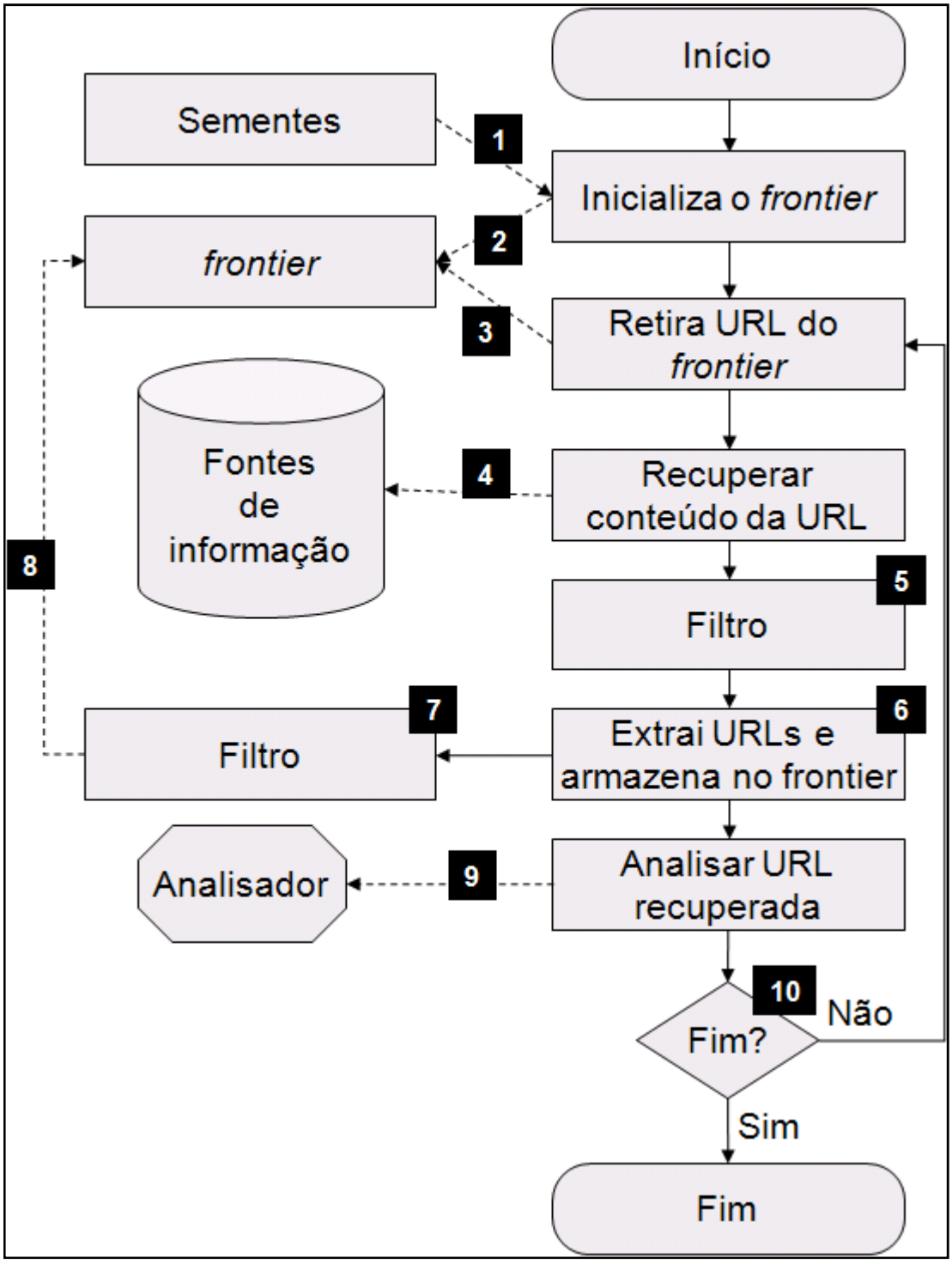


Figura 3. Arquitetura geral para a coleta automática de dados.

Fonte: Adaptado de Liu (2009, p.275).

A arquitetura tem apreço pela modularidade, ou seja, é possível instanciá-la através de um software de maneira a tolerar a substituição dos seus componentes e assim atacar uma variedade de necessidades de coleta. Pode-se, por exemplo, agregar módulos responsáveis por recuperar o conteúdo de URLs, um para cada tipo de fonte de informação. Assim, caso o

usuário tenha a pretensão de alcançar dados disponíveis em repositórios FTP, sendo esta fonte não suportada nativamente pelo coletor, é possível estendê-lo sem prejudicar o software existente pela configuração de um novo módulo.

Uma arquitetura flexível, com apego pela modularidade, também necessita suprir os elementos necessários para que o hiato entre as necessidades de coleta do usuário e os dados seja transposto. Destarte, é preciso que ela provenha os meios para compreender tais necessidades, transferindo essa compreensão para o sistema.

A arquitetura captura a intenção dos usuários através das sementes ou URLs gênese, dos filtros, analisadores e também do *frontier*. A relação entre esses quatro componentes é suficientemente prestigiosa para permitir que uma ampla gama de necessidades de coleta possa ser computacionalmente abstraída.

As sementes (ou URLs gênese) formam o marco inicial, o ponto de partida para o processo de coleta. As URLs identificarão não somente a fonte de dados, mas também o recurso dentro dela por onde iniciar o processo de coleta. Por exemplo, a semente http://www.brasil.gov.br/noticias/ identifica a Web como fonte de informação e aponta o portal de notícias do governo brasileiro como a porção da fonte contendo dados relevantes.

Também dentro do processo de coleta, os filtros são fundamentais para capturar as necessidades dos usuários. Se os dados relevantes estiverem todos contidos dentro da seção de notícias do portal do governo brasileiro na Internet, esse precisará informar tal desejo ao coletor. Caso contrário, o agente computacional varrerá toda a Web a partir da semente. Filtros podem ser vistos como restrições impostas ao processo de coleta. Por exemplo, é possível restringir o coletor à recuperação de URLs filhas da URL gênese ou a arquivos de certo tipo (e.g., imagens e apresentações do Power Point).

Os analisadores também são essenciais para o entendimento das necessidades dos usuários. Eles fornecem o ponto de contato entre aplicações consumidoras de dados coletados e o coletor. Deste modo, para cada documento coletado, os analisadores registrados com o coletor são notificados, sendo eles os responsáveis por dar um destino útil aos dados. Nesse ponto, é possível decidir sobre a análise ou não de determinado documento coletado, além da maneira como a análise será efetuada.

Por último, muitas das necessidades dos usuários de um serviço de coleta podem estar imbuídas no *frontier*, que é a lista mantenedora das URLs por serem coletadas. Heurísticas para a ordem com que as URLs são mantidas, ou para controlar tanto o processo de adição quanto o de remoção de URLs, podem ser construídas. Tais heurísticas são úteis para determinar, por exemplo, que as URLs mais importantes, segundo alguma estimativa, sejam coletadas primeiro. Essas heurísticas também podem ser utilizadas para que URLs referenciando um mesmo *host* sejam desempilhadas uma por vez, reduzindo assim a sobrecarga sobre os computadores provedores de dados, ou ainda, que URLs já coletadas não voltem ao componente, evitando que elas sejam processadas repetidas vezes.

2.2 Flexibilizando a Arquitetura

A habilidade da arquitetura em lidar com aspectos específicos ao processo de coleta é o evento que permitirá a sua maleabilidade. Tal plasticidade da arquitetura é imprescindível para que uma fração considerável das necessidades dos usuários possa ser atendida, tornando-a apta à resolução de diversos problemas de coleta. Sendo assim, é necessário identificar tais aspectos, perscrutá-los e redesenhar a arquitetura genérica com a suplementação deles.

A literatura acadêmica envolvendo coletores descreve aspectos que vão de encontro à criação de uma arquitetura flexível. Os que serão abordados pelo presente trabalho são a coleta focada, coleta eficiente, coleta distribuída, polidez, sincronização dos dados, desempenho, tolerância à falha, entendimento de múltiplos protocolos e tipos de documentos, e armadilhas presentes nos ambientes provedores de dados. Cada aspecto será retratado nas seções que seguem.

2.2.1 Coleta Focada

Muitas das necessidades de coleta dos usuários são por documentos relevantes a um determinado tópico (LEE *et al*., 2008). Por exemplo, viajantes provavelmente demandarão páginas Web referentes ao local que desejam visitar. Da mesma forma, economistas podem estar interessados em notícias sobre a bolsa de valores, assim como acadêmicos por artigos relacionados à pesquisa que estejam efetuando. O mecanismo que operacionaliza o coletor para que esse encontre dados acerca de determinado tópico é chamado de coleta focada.

No contexto da Web, onde tal prática é comumente estudada, a coleta focada ainda busca a resolução de outros problemas. Por conta da grande e crescente quantidade de dados

espalhados em tal meio, coletar todos os documentos impõe restrições que poucos coletores estão aptos a transpassar. Deste modo, é possível perceber a coleta focada como uma alternativa viável para a busca por dados em ambientes contendo muita informação.

A literatura sobre coleta focada é significativa. Aggarwal, Al-Garawi e Yu (2001) apresentam uma proposta para coleta focada baseada na aprendizagem da estrutura de relacionamentos entre documentos durante o processo de recuperação. Chakrabarti, Punera e Subramanyam (2002) demonstram a quantidade de informação que *hyperlinks* contêm sobre determinado tópico. Pant, Srinivasan e Menczer (2002) discorrem sobre a diferença entre iniciar o processo de coleta focada a partir de um conjunto relevante de páginas Web e a partir de páginas que estejam a poucos links de distância de documentos relevantes. Angkawattanawit e Rungsawang (2002) abordam o processo de coleta focada em sua segunda iteração, ou seja, após um primeiro procedimento de coleta já ter ocorrido. Stamatakis *et al*. (2003) apresentam CROSSMARC, um coletor focado na busca por páginas Web relevantes a um determinado domínio. Li, Furuse e Yamaguchi (2005) expõem a utilização de árvores de decisão no processo de coleta focada. Gao, Lee e Miao (2006) propõem uma estratégia para a coleta focada em documentos provenientes de uma determinada região geográfica. Por fim, estudos relacionados à inserção de ontologias no processo de coleta focada são descritos pelos seguintes autores: Ehrig e Maedche (2003); Su *et al*. (2005); Kozanidis (2008); Pahal, Chauhan e Sharma (2007); Dong, Hussain, Chang (2008), Zheng, Kang e Kim (2008), Luong, Gauch e Wang (2009) e Dong, Hussain e Chang (2009).

Existem variadas formas para que a coleta focada seja inserida na arquitetura genérica, sendo uma delas através do *frontier*, como mostra a Figura 4. Nela são apresentadas duas listas. A primeira armazena URLs com alta probabilidade de conterem conteúdo relevante a um determinado tópico, e a segunda mantém a lista URLs onde a probabilidade é baixa. Sempre que uma URL for coletada e atestada como relavante, as URLs filhas dessa são enviadas à lista de URLs com probabilidade de serem também relevantes, o que não acontece com URLs filhas de uma não importante, que vão para a lista de URLs marcadas como possivelmente irrelevantes no *frontier*. Durante a remoção de uma URL do *frontier*, é concedida maior prioridade para a retirada de URLs com maior probabilidade de serem relevantes. As URLs não relevantes são descartadas pelo analisador (Figura 3, passo 9), garantindo que o usuário só receberá documentos relevantes ao tópico de interesse.

Esse mecanismo garante que o coletor dedique a maior parte de seu tempo recuperando URLs com alta probabilidade de serem relevantes a um determinado tópico, porém sem deixar de visitar URLs provavelmente irrelevantes que podem levar a documentos importantes.

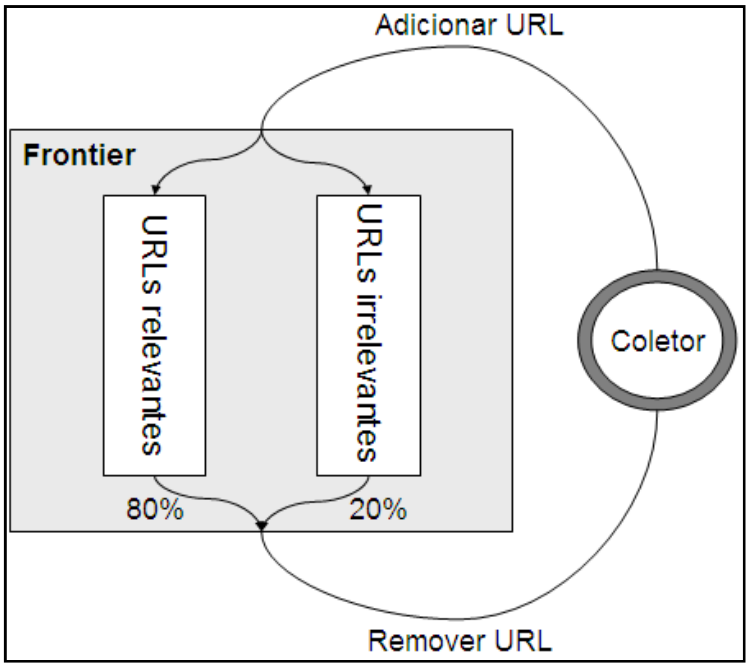


Figura 4. *Frontier* configurado para a coleta focada.

A marcação de uma URL como relevante ou não é fruto de heurísticas como a análise do conteúdo das URLs coletadas e a inspeção das âncoras nas URLs por serem recuperadas, para ilustrar duas. A análise do conteúdo de uma URL coletada permite identificar se ela é ou não relevante para o cliente do sistema coletor (WANG, 2009). Caso seja, pode-se atribuir uma relevância maior às URLs filhas dessa do que às URLs filhas de uma URL que teve seu conteúdo inspecionado e foi declarada não relevante. Já o exame das âncoras nas URLs admite a identificação de URLs relevantes se o texto da âncora contiver termos que sejam do interesse do cliente do sistema coletor (YADAV; SHARMA; GUPTA, 2009).

2.2.2 Coleta Eficiente

Coleta eficiente é aquela que, além de recuperar documentos relevantes às necessidades do usuário, aborda tais documentos durante os estágios iniciais do processo de coleta. Esse tema é exaustivamente tratado por Cho, Garcia-Molina e Page (1998), que apresentam um estudo sobre a coleta eficiente através da ordenação de URLs, propondo heurísticas que demonstram como um coletor deve selecionar as URLs a partir do *frontier*.

Diferentemente da coleta focada, que entrega aos usuários dados relacionados a determinado tópico, a coleta eficiente, grande parte das vezes, recuperará todos os documentos de determinada fonte, priorizando a coleta dos mais relevantes nos momentos iniciais da tarefa. Por exemplo, algum período após a coleta de um conjunto de documentos, o coletor poderá revisitá-los em busca de mudanças. Caso o número de documentos seja significativo e a atualização desses uma necessidade, é essencial que o coletor revisite primeiro os documentos modificados, aqui os mais relevantes, apurando os demais posteriormente.

A integração da coleta eficiente pela arquitetura se dá pela ordenação das URLs dentro do *frontier*, com as mais relevantes situadas no começo da fila e as menos ao final. Cho, Garcia-Molila e Page (1998) aplicaram o método à coleta de páginas Web utilizando quatro abordagens para a ordenação dos elementos do frontier: *PageRank*, *backlink count*, *breadth-first* e um algoritmo randômico.

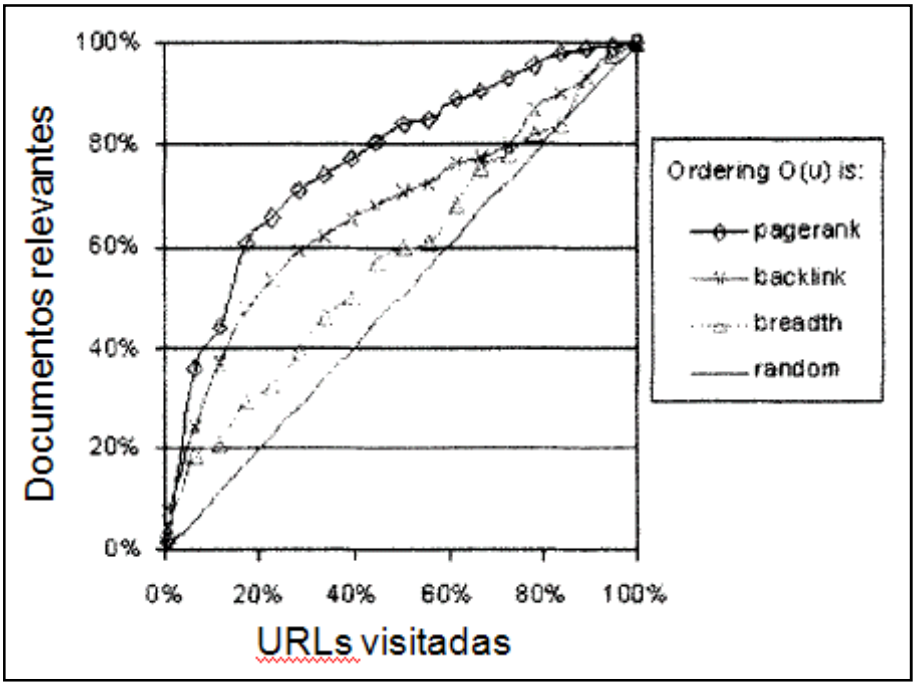


Figura 5. Arquitetura geral para a coleta automática de dados.
Fonte: Adaptado de Cho, Garcia-Molina e Page (1998).

Os resultados são apresentados na Figura 5, onde o eixo horizontal denota o percentual de URLs visitadas e o vertical o percentual de documentos relevantes coletados. Na figura é possível notar que, ao final do processo de coleta, todos os documentos são recuperados. A diferença está no percentual de visita a documentos relevantes no estágio inicial da coleta.

Nota-se que o melhor desempenho se dá através do uso do *PageRank*, onde o coletor consegue, com apenas 20% das URLs visitadas, recuperar pouco mais de 60% dos documentos relevantes.

2.2.3 Coleta Distribuída

Um coletor que anseie pela recuperação de grandes quantidades de dados, mesmo que extremamente rápido, não conseguirá a partir de uma única máquina realizar o processo de coleta de grandes fontes de dados em tempo hábil. Esse é o problema enfrentado por coletores que objetivam a recuperação de grandes porções da Web, ou ainda, coletores que necessitam se esgueirar por grandes Intranets como as encontradas em muitas corporações. Em ambos os casos é indispensável que o domínio da coleta seja particionado e executado em diferentes computadores.

Uma das tarefas essenciais à concepção de coletores distribuídos é a aplicação de boas funções de particionamento. Elas determinam a maneira como as URLs serão distribuídas entre os computadores responsáveis pela coleta. Cho e Garcia-Molina (2002) dividem a função de particionamento em três categorias: independente, dinâmica e associação estática.

Na categoria independente, cada processo executa em determinado computador sem trocas com outros processos. Esta categoria, apesar de não apresentar grandes desafios tecnológicos, permite que uma mesma URL possa ser recuperada mais de uma vez por dois ou mais processos. Já a dinâmica existe quando um coordenador central divide logicamente o domínio de coleta associando cada URL a um determinado processo. E a associação estática, do mesmo modo que a dinâmica, distribui URLs em processos, porém a maneira como a associação ocorre é definida antes do início da coleta.

Cada processo é responsável por uma fração dos documentos a serem coletados, aqui chamada de partição. Entretanto, alguns documentos coletados por um determinado processo podem conter URLs referentes a outras partições. Nesse caso, o coletor pode operar sob alguns modos específicos, como o *firewall*, sobrescrita e troca, todos descritos por Cho e Garcia-Molina (2002).

No modo *firewall*, cada processo recupera somente URLs que pertençam a sua partição, descartando as outras URLs. Apesar deste modo assegurar que uma mesma URL não seja coletada por múltiplos processos, alguns documentos podem não ser recuperados, já

que existem casos onde URLs somente serão alcançadas através do relacionamento entre as partições, como mostra a Figura 6. Nela são apresentados dois processos, com cada um envolto por linhas pontilhadas onde as letras representam URLs. Note que o coletor da partição esquerda só é capaz de alcançar as URLs *d* e *e* caso algum tipo de comunicação ocorra entre as partições, o que não acontece no modo *firewall*.

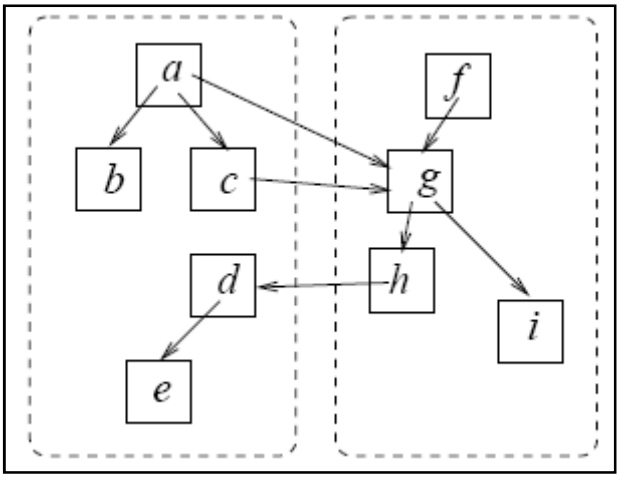


Figura 6. Processos de coleta atuando em diferentes partições.
Fonte: Cho e Garcia-Molina (2002).

O modo sobrescrita realiza inicialmente a coleta de URLs pertencentes à sua partição. Quando as URLs acabam, ao contrário do modo *firewall*, que não segue URLs de outras partições, o modo sobrescrita coleta outras URLs até encontrar alguma referente à sua partição, retornando a ela. Na Figura 6, caso o modo sobrescrita seja empregado, após o processo da esquerda coletar as URLs *a*, *b* e *c*, ele seguirá para as URLs *g* e *h*, retornando para a sua partição e coletando as URLs *d* e *e*. Nesse modo, uma URL pode ser recuperada por mais de um processo, gerando duplicidades na coleta. Da mesma maneira que o modo *firewall*, aqui não existe a necessidade de comunicação entre os processos.

O modo troca, também descrito por Cho e Garcia-Molina (2002), é marcado pelo câmbio constante de URLs entre os processos de coleta, onde cada um não coleta URLs referentes a outras partições. Na Figura 6, o processo coletando a partição da esquerda, logo após recuperar a URL *c*, enviará a URL *g* para ser recuperada pelo processo responsável pela partição da direita. O processo da esquerda entrará então em modo de espera até que receba a URL *d* pelo processo da direita. Ao contrário dos modos *firewall* e sobrescrita, o modo troca impõe uma comunicação excessiva entre os processos que deve ser minimizada.

Existem algumas alternativas para a minimização da carga imposta pela troca de URLs entre processos dentre as quais ganham destaque a comunicação em lotes e a replicação, ambas descritas por Cho e Garcia-Molina (2002). A comunicação em lotes, ao invés de submeter URLs às outras partições no momento que as encontra, armazena um conjunto delas e as envia em um único momento. Esta abordagem melhora a comunicação entre os processos já que todas as URLs são enviadas de uma única vez através de uma única conexão. Além disso, caso determinada URL apareça duas ou mais vezes, somente uma será submetida. Já a replicação mantém uma cópia das URLs mais importantes em cada um dos processos, eliminando a necessidade de troca dessas URLs. As URLs mais importantes podem ser definidas através de uma coleta prévia.

Há ainda a incógnita quanto ao particionamento da Web, ou outra fonte de informação, entre os processos integrantes de um sistema distribuído. Apesar das funções de particionamento abordadas nesta seção focarem a cisão da Web, elas se adaptam à segmentação de qualquer outra fonte com pouco esforço. São descritas por Cho e Garcia-Molina (2002) as funções de particionamento baseadas na codificação de URLs e *sites*, além de funções fundamentadas na divisão hierárquica da Web.

As funções de particionamento apoiadas na codificação de URLs e *sites* são muito semelhantes. No primeiro caso, cada URL é enviada através de uma função *hash* que retorna o valor utilizado na identificação do processo responsável pela coleta do recurso. Nesse esquema, páginas de um mesmo *site* poderão ser coletadas por processos distintos. No segundo caso, somente a porção da URL contendo o nome do *site* é utilizada na computação do código *hash* e, sendo assim, todas as páginas pertencentes a um mesmo *site* Web serão coletadas por um único processo.

Já o particionamento hierárquico decompõe a Web por domínio. Por exemplo, todas as URLs pertencentes ao domínio *.com* serão coletadas por determinado processo, as URLs do domínio *.net* por outro e assim sucessivamente. É possível ainda utilizar o particionamento hierárquico para dividir a Web por língua ou país (e.g., ".pt" e ".br" para páginas em português e somente ".br" para *sites* do Brasil). Esta manobra é especialmente interessante quando existem processos geograficamente distribuídos, como um processo no Brasil coletando todas as páginas da América do Sul, enquanto outro no Japão lida com a Web asiática.

Wan e Tong (2008) salientam em seus estudos sobre funções *hash* para a divisão da Web que uma boa função de particionamento é necessária para descentralizar o processamento de um coletor e garantir o balanceamento da carga do sistema. A função precisa ser suficientemente engenhosa para explorar a estrutura de *links* entre os documentos de maneira eficiente, mantendo igualitário o volume de trabalho entre os processos. Deste modo, é interessante que os coletores provenham uma maleabilidade tanto na escolha de funções *hash* quanto na seleção de algoritmos para a divisão hierárquica.

Os elementos descritos nesta seção focaram aspectos genéricos, essenciais à concepção de arquiteturas para a coleta distribuída. Muitos deles estão materializados em coletores e descritos pela literatura acadêmica, como é evidenciado por Kritikopoulos, Sideri e Stroggilos (2004) em um inventivo coletor baseado em serviços Web, ou nos escritos de Shkapenyuk e Suel (2002) sobre um coletor distribuído e extremamente eficiente. Outros coletores impregnados por tais aspectos são descritos por Boldi *et al*. (2004) e Zhu *et al*. (2008).

2.2.4 Polidez

Coletores são potenciais extenuadores de recursos preciosos como banda de Internet, memória e ciclos de CPU (*Central Processing Unit*), tanto dos computadores e rede onde o processo de coleta executa quanto das máquinas e redes de onde os dados são servidos. Por serem muitas vezes escassos, fazer bom uso desses recursos é uma preocupação constante durante o desenvolvimento de coletores que devem mitigar constantemente os riscos inerentes ao consumo demasiado de tais bens.

Não é incomum o relato de testes de coletores que, de alguma maneira, afetaram negativamente os recursos que operavam. Pela carga imposta por muitos coletores, foram observados colapsos em sistemas de *firewall* e servidores Web (HAFRI; DJERABA, 2004), percebidos gargalos na utilização de ciclos de CPU, acesso à memória principal e ao disco, na capacidade de armazenamento e durante operações de entrada e saída na rede (BRIN; PAGE, 1998), além de constatada uma degradação no desempenho da rede onde o coletor foi instrumentado, forçando o controle da velocidade com que os recursos eram recuperados para minimizar o impacto em outros usuários (SHKAPENYUK; SUEL, 2002).

Além da perturbação causada ao computador e rede operados pelo coletor, abalos semelhantes podem ocorrer junto aos computadores servidores de dados. Cada requisição a

um recurso pelo coletor, realizada através de uma URL, implica ciclos de CPU e banda de Internet consumidos pelo servidor para o envio de uma resposta. Ou seja, quanto maior o número de requisições partindo do coletor, mais ciclos de CPU e banda são utilizados pelo computador servidor de dados.

Servidores podem repelir coletores através do padrão *Robot Exclusion* (KOSTER, 1994), que informa quais coletores podem recuperar quais documentos. Assim, computadores servidores despendem menos recursos com coletores, salvando esses para outros usuários. Outro modo de preservar recursos acontece quando os coletores são polidos, ou seja, quando é contemplada uma quantidade significativa de tempo entre requisições para um mesmo computador servidor de dados. Sobre tal quantidade de tempo, Silva *et al*. postulam que

> Uma requisição a um servidor HTTP deve somente ser encaminhada se o servidor estiver livre. Um servidor é considerado livre se pelo menos 60 segundos tiverem passado desde a resposta para a última requisição (enviada para este servidor) (SILVA *et al*., 1999).

Por fim, o esforço dos coletores na busca por polidez pode ser facilitado através da concepção de servidores de dados que provenham, além dos documentos solicitados, um conjunto de metadados acerca desses (BRANDMAN *et al*., 2000). Através da exportação de metadados pelos servidores é possível perceber dois momentos onde os coletores podem ser mais polidos: durante a atualização de documentos já coletados e no decorrer de uma coleta focada em diferentes tipos de mídia.

No primeiro caso, o da atualização dos documentos sem a utilização de metadados, o coletor necessita recuperar novamente todos os arquivos, analisar o conteúdo de cada um e decidir se devem ou não sofrer atualização. Este procedimento implica um elevado tráfico de dados do servidor para o coletor, algo especialmente maléfico quando somente alguns poucos arquivos demandam atualização. Aqui, uma estratégia baseada em metadados se desenrola pela indagação do coletor ao servidor pelos arquivos modificados desde sua última visita. Consequentemente, um conjunto de metadados informando quais documentos foram modificados é entregue pelo servidor, iniciando a recuperação desses pelo coletor e poupando banda de Internet e ciclos de CPU, tanto no lado de quem coleta quanto da parte que serve dados.

A proposta para o uso de metadados na coleta focada em diferentes tipos de documentos é semelhante à atualização dos documentos. Por exemplo, a solução corrente para

a coleta de todos as imagens contidas em um determinado site é navegá-lo por completo, visitando outros tipos de arquivos, especialmente páginas HTML, até que todas as imagens sejam recuperadas. A abordagem pelos metadados é mais direta. Basta solicitar ao servidor a lista de URLs para as imagens do site e coletá-las individualmente, poupando banda de Internet e ciclos de CPU pelas partes envolvidas no processo de coleta.

Para que tanto o caso da atualização dos documentos como o da coleta focada seja possível, é imprescindível que o tamanho do metadado trocado entre o coletor e servidor seja pequeno. Brandman *et al*. (2000) explicam que, para o armazenamento da data da última atualização de determinado documento e sua URL, são necessários, na maioria dos sistemas de arquivos, 64 bytes. A compressão pode levar à redução deste número por um fator de três. Isso significa que 100Kb podem armazenar metadados para aproximadamente 5000 páginas Web, ou um site de tamanho considerável, demonstrando um custo modesto para a exportação de metadados.

2.2.5 Sincronização

Sincronização refere-se à manutenção da base de documentos recuperados atualizada face às mudanças ocorridas nos dados armazenados pelas fontes utilizadas na coleta (CHO; GARCIA-MOLINA, 1999). A Figura 7 captura tal definição. Nela existem dois espaços contendo dados, onde um representa a fonte de informação sendo coletada e o outro a cópia local com os dados coletados. A fonte de informação é atualizada independentemente e não notifica possíveis interessados (aqui o coletor) sobre eventuais mudanças como, por exemplo, a adição ou remoção de páginas Web. Nesse caso, o coletor inspecionará continuamente a fonte de informação por mudanças, uma atividade nomeada *pooling*, atualizando a base local quando elas ocorrerem. Nessa interação com a fonte de informação, o coletor precisa aprender o padrão de mudanças para inspecioná-la mais eficaz e eficientemente.

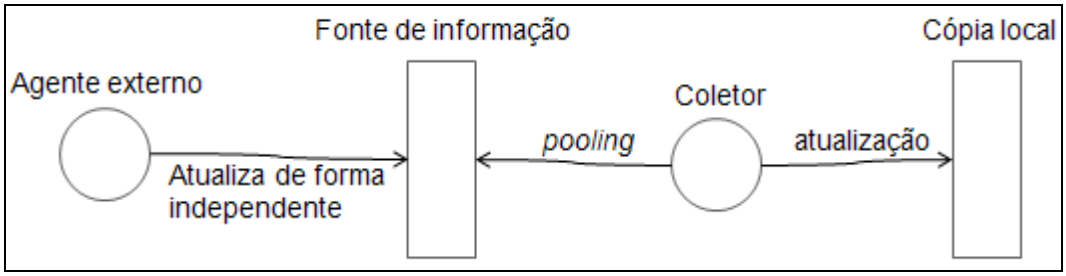


Figura 7. Processo de sincronização.

Fonte: Adaptada de Cho e Garcia-Molina (2000).

Um coletor que sincronize os dados de modo eficiente e eficaz é necessário, por exemplo, quando grandes fontes de informação como a Web são manuseadas. Cho e Garcia-Molina dizem que

> Um Web *crawler* é um programa que automaticamente visita páginas Web e constrói cópias locais ou índices dessas páginas. Para manter a cópia local ou índice atualizados, o *crawler* revisita periodicamente as páginas e as atualiza junto à cópia local. Um *crawler* comum usualmente revisita todo um conjunto de páginas periodicamente, atualizando todas elas. Entretanto, se o *crawler* pode estimar com que frequência uma determinada página muda, ele pode revisitar somente as páginas que mudaram (com alta probabilidade) e melhorar o coeficiente de atualização da cópia local sem consumir demasiada banda de Internet. (CHO; GARCIA-MOLINA, 2003b).

Além da coleta de grandes massas de dados da Web, não é incomum encontrar casos onde a resolução deste problema constitui um dos alicerces de determinado tipo de sistema. Por exemplo, agregadores de notícias, que coletam e disponibilizam conteúdo eletrônico publicado por jornais, *blogs* e fontes correlatas, por lidarem com um tipo de informação efêmera, muitas vezes relacionada a eventos correntes no mundo, precisam divulgá-las tão logo surjam, forjando um belo exercício de sincronização.

Outros exemplos podem ser relacionados, como buscadores que realizam periodicamente eventos de sincronização para não surpreender seus usuários com *links* quebrados, ou ainda, deixar de apresentá-los informação relevante, e também *data warehouses* com ciclos de coleta que, além de manterem o sincronismo dos dados, conservam o produto da análise consistente com o ambiente externo.

O tema sincronização é substancialmente abordado pela literatura acadêmica. Cho e Garcia-Molina (1999) apresentam algoritmos para sincronização de dados contidos em fontes de dados autônomas, delineando respostas a questões sobre a frequência com que tais sincronizações precisam ser realizadas para que um determinado percentual da cópia local dos documentos se mantenha atualizada. Cho e Garcia-Molina (2000, 2003a, 2003b) também exploram o processo de sincronização integrado ao ato da coleta e não como uma atividade isolada que é geralmente executada em modo *batch*. Estudos semelhantes são realizados por Edwards, McCurley e Tomlin (2001) e Cho e Ntoulas (2002). No contexto de notícias, a sincronização desse tipo de informação é exaurida por Sia, Cho e Cho (2007).

Para ilustrar a técnica, a proposta para a sincronização descrita por Cho e Garcia-Molina (2000) será descrita. Nela, inicialmente são apresentados os conceitos de *freshness* e

age. Dado uma cópia local $S = \{e_1,...,e_n\}$ contendo N elementos (e.g. páginas Web), idealmente todos os elementos estarão atualizados face à fonte de informação de onde originaram, porém somente M tal que $M < N$ elementos, na grande maioria dos casos, estarão sincronizados em um ponto específico no tempo. Sendo assim, o *freshness* de S em um dado ponto no tempo t é capturado como $F(S;t) = M/N$. Deste modo, $F(S;t)$ será igual a 1 se todos os elementos estiverem sincronizados e 0 caso todos estejam desatualizados. A equação pode ser reformulada para:

$$F(S;t) = \frac{1}{N}\sum_{i=1}^{N} F(e_i;t). \qquad \text{Equação 1}$$

Onde $F(e_i;t)$ é dado por

$$F(e_i;t) = \begin{cases} 1, \text{se } e_i \text{ esta atualizado no tempo } t \\ 0, \text{caso contrario.} \end{cases} \qquad \text{Equação 2}$$

O conceito *age* define o quão antiga uma base de dados está, e pode ser descrito através da seguinte equação:

$$A(S;t) = \frac{1}{N}\sum_{i=1}^{N} A(e_i;t). \qquad \text{Equação 3}$$

Onde $A(e_i;t)$ é dado por

$$A(e_i;t) = \begin{cases} 0, \text{se } e_i \text{ esta atualizado no tempo } t \\ t - tempo\ modificacao\ de\ e_i, \text{caso contrario.} \end{cases} \qquad \text{Equação 4}$$

Também é necessária a definição do tempo médio do *freshness* de um elemento e_i, dado por $\bar{F}(e_i)$, bem como o tempo médio do *freshness* de toda a base coletada S, $\bar{F}(S)$.

$$\bar{F}(e_i) = \lim_{n\to\infty} \frac{1}{t} \int_0^t F(e_i; t)dt. \qquad \text{Equação 5}$$

$$\bar{F}(S) = \lim_{n\to\infty} \frac{1}{t} \int_0^t F(S; t)dt\,. \qquad \text{Equação 6}$$

Para construir um sistema de sincronização eficaz, é preciso saber como os elementos do mundo real mudam. Cho e Garcia-Molina assumem que os elementos, no exemplo deles páginas Web, são modificados através de um processo de *Poisson*, usualmente utilizado para modelar sequências de eventos que acontecem independente e aleatoriamente, porém a uma taxa fixa no tempo. Através do processo de *Poisson* é possível analisar o *freshness* e *age* de um elemento no tempo. Mais precisamente, é possível computar o valor esperado do *freshness* e *age* de e_i em *t*. Deste modo, o *freshness* esperado é dado por

$$E[F(e_i; t)] = 0\,.\left(1 - e^{-\lambda t}\right) + 1\,.\,e^{-\lambda t} = e^{-\lambda t}. \qquad \text{Equação 7}$$

Note que o *freshness* esperado é 1 no tempo $t = 0$ e aproxima de 0 com o passar do tempo. Semelhantemente, é possível obter o valor esperado de *age*. Nesse caso, o valor é 0 no tempo $t = 0$ e aproximadamente igual a *t* com o passar do tempo, como mostra a equação 8. Os cálculos completos foram omitidos e podem ser encontrados em Cho e Garcia-Molina (2000).

$$E[A(e_i; t)] = \int_0^t (t - s)\left(\lambda e^{-\lambda s}\right)ds = t(1 - \frac{1 - e^{-\lambda t}}{\lambda t}). \qquad \text{Equação 8}$$

Dado o *framework* matemático, é necessário delinear os passos para a sincronização, ou seja, como a base local pode ser atualizada. Primeiramente é necessário dividir o tempo total em unidades de tempo *I*. A cada unidade de tempo, *N* elementos serão sincronizados. Variando o valor de *I* é possível ajustar com que frequência a base local é atualizada. O valor de *N* também pode ser ajustado.

Mesmo após a escolha do número de elementos a serem sincronizados por unidade de tempo *I*, ainda é preciso decidir com que frequência cada elemento será atualizado

individualmente. Por exemplo, caso existam três elementos, e_1, e_2 e e_3, e sabe-se que cada um muda com frequência igual a $\lambda_1 = 4$, $\lambda_2 = 3$ e $\lambda_3 = 2$ (vezes/dia), foi decidido sincronizar a base local a uma taxa total de 9 elementos/dia (I = 1, N = 9). Assim, é possível sincronizar os elementos uniformemente, cada um 3 vezes ao dia, ou sincronizar os elementos proporcionalmente: $f_1 = 4$, $f_2 = 3$ e $f_3 = 2$ (vezes/dia).

Também é necessário definir com que ordem os elementos serão sincronizados dentro do intervalo *I* de tempo. É possível seguir uma ordem fixa, iterando sobre todos os elementos sempre da mesma maneira, variar a ordem de acordo com alguma medida, ou iterar sobre os elementos aleatoriamente. O cálculo de *I* e de *N*, bem como a decisão de quais elementos serão sincronizados e em qual ordem, ambos omitidos, porém descritos em Cho e Garcia-Molina (2000), são realizados utilizando as equações apresentadas para o cálculo do *freshness* e *age*.

2.2.6 Desempenho

Um dos pilares fundamentais à construção de coletores é o desempenho. Raro é encontrar alguma outra característica que esteja tão presente em cada componente, cada detalhe técnico que compõe esse tipo de sistema. Esta preocupação advém do contexto onde os coletores operam. São geralmente ambientes estufados por milhões de documentos, muitos dos quais fisicamente distantes como usualmente ocorre na Web. Neste cenário, cada ajuste, cada *byte* poupado incrementa perceptivelmente a velocidade com que os documentos são recuperados, melhorando o desempenho do coletor.

O desempenho é importante em determinados contextos. Recuperar alguns poucos documentos da máquina local pode afrouxar tal requisito, ao passo que coletar milhões de documentos da Web, mantendo a base coletada atualizada, traz à superfície requisitos únicos quanto ao desempenho do sistema de coleta.

Com o surgimento de fontes de dados cada vez maiores, contendo grandes massas de dados, é imperativo realizar tanto a coleta dos dados como a atualização dos dados coletados em um espaço de tempo adequado. O problema é acentuado quando diversas fontes de dados são inspecionadas simultaneamente pelo processo de coleta, como na busca por dados na Web e, concomitantemente, em uma grande Intranet corporativa.

Os recursos que podem ser empregados para melhorar o desempenho de um coletor serão apresentados até o final desta seção. Primeiramente, serão evidenciadas as fontes de latência inerentes à recuperação de dados da Web. A seguir, será abordado o emprego de múltiplas *threads* e requisições assíncronas pelo processo de coleta. A normalização de URLs também exerce papel importante no desempenho de um coletor, assim como testes para saber se determinado conteúdo ou URL já foram coletados e também a lida com *hosts* duplicados, tópicos esses que também serão apresentados.

Fontes de Latência na Web

Habib e Abrams (2000) analisam fontes de latência durante o processo de recuperação de páginas Web, mensurando essas através de diversos cenários como com o cliente e servidor fisicamente alocados tanto no mesmo país quanto em países diferentes. As fontes de latência apontadas pelo estudo são a resolução de *hosts* em IPs (*Internet Protocol*), o tempo para o estabelecimento de uma conexão TCP (*Transmission Control Protocol*) entre o cliente e servidor, o tempo para que, depois de estabelecida a conexão TCP entre cliente e servidor, o primeiro *byte* seja recuperado e o tempo para o *download* completo de um arquivo. Um coletor que almeje desempenhar eficientemente suas atribuições deve compreender cada uma dessas fontes e prover mecanismos para que a latência gerada a partir delas seja mitigada.

É comum o relato pela literatura que um dos gargalos durante o processo de coleta é a resolução de *hosts* em IPs. Heydon e Najork (1999), utilizando um componente síncrono para esta tarefa, experimentaram 87% do tempo total da coleta despendido nesta atividade. Após ajustes no software, sendo os mais notáveis a construção de um mecanismo assíncrono e também o emprego de uma estratégia de *cache* dos dados, o percentual foi reduzido para 25%. Hafri e Djeraba (2004) reportaram uma melhora substancial no desempenho do coletor por resolverem os *hosts* em IPs antes do processo de coleta. Shkapenyuk e Suel (2002) identificaram o processo de resolução de *hosts* em IPs como um dos gargalos no processo de coleta pelo fato desse gerar um número significativo de quadros, restringindo a velocidade do coletor por conta da limitada capacidade de processamento de quadros pelo roteador, um Cisco 3620. Buswell (2003) também reportou a resolução de *hosts* em IPs como um dos processamentos que consomem uma significativa quantidade de tempo durante o ato da coleta, melhorando o desempenho do coletor através do uso de requisições assíncronas. Já Habib e Abrams (2000) atribuíram ao processo de resolução de hosts a generosa fatia de 33% do tempo total necessário à recuperação de um documento da Web.

Outra fonte de latência ocorre entre o contato inicial do cliente com o servidor e o recebimento do primeiro *byte* relativo ao documento solicitado. Essa etapa é dividida em duas, sendo a primeira o estabelecimento de uma conexão TCP entre o cliente e servidor, seguida pelo tempo necessário para o servidor processar a requisição do cliente e enviar o primeiro *byte* do arquivo solicitado.

O protocolo TCP contribui, segundo Habib e Abrams (2000), substancialmente para a degradação do desempenho no contato inicial entre o cliente e servidor, já que é necessário um *3-way handshake*[6] para o estabelecimento de uma conexão. O problema é agravado quando o tamanho dos documentos transmitidos é sensivelmente pequeno, com o *handshake* representando um percentual significativo do total de *bytes* transferidos durante a requisição. Para mitigar essa fonte de latência, uma das técnicas utilizadas consiste em manter a conexão TCP persistente. Ela será compartilhada por várias requisições HTTP e eliminará a necessidade de múltiplos *handshakes*.

Além do TCP, é possível que situações adversas relacionadas ao computador servidor de dados também impactem diretamente o tempo para o retorno do primeiro *byte*, retardando o envio por uma enormidade de causas que vão do extravio de ciclos de CPU por conta do atendimento a múltiplas requisições até problemas relacionados ao hardware sobre o qual o servidor executa.

Esse aspecto foi explorado por Habib e Abrams (2000), que sobrecarregaram um servidor com 30 requisições por segundo. Apesar do esforço, foi notado o acréscimo de apenas três milissegundos ao tempo necessário para o estabelecimento de uma conexão TCP e o envio do primeiro *byte* relacionado ao documento requisitado. O tempo total dessa ação foi de 3,58 segundos.

A quantidade representativa de tempo para a percepção do primeiro *byte* pelo cliente (3,58 segundos) foi explicada por Habib e Abrams (2000) como sendo produto de hardwares, especialmente roteadores, com limitada capacidade para gerir grandes quantidades de quadros e também a baixa velocidade de acesso à Internet, que impacta diretamente no desempenho do coletor.

---

[6] http://en.wikipedia.org/wiki/Transmission_Control_Protocol

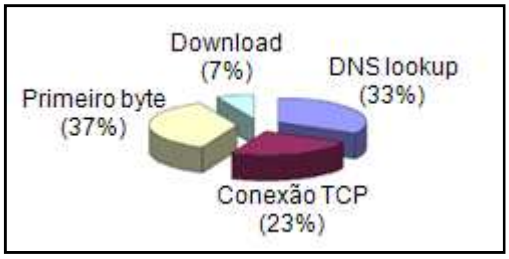


Figura 8. Distribuição do tempo para o acesso a uma URL.
Fonte: Adaptada de Habib e Abrams (2002).

A Figura 8 mostra um comparativo entre as quatro fontes de latência definidas no início desta seção: resolução de *hosts* em IPs (na figura, DNS *lookup*), o estabelecimento de uma conexão TCP, o tempo necessário para o servidor interpretar a requisição HTTP e oferecer ao cliente o primeiro *byte* do documento solicitado e também o tempo para o *download* de todos os *bytes*. Cada etapa acompanha o percentual do consumo de tempo total para a coleta de um documento.

Múltiplas Threads e Requisições Assíncronas

Ao contrário de coletores distribuídos, que fazem uso de diversos computadores para uma melhora no desempenho, coletores paralelos residem em somente um computador, explorando o poder de processamento oriundo de múltiplas CPUs. Existem duas técnicas pelas quais é possível a realização de coletores paralelos. A primeira prevê o emprego de múltiplas *thread*s, onde cada uma é encarregada da coleta e processamento de um único documento. A segunda é através do uso de conexões de rede assíncronas.

A estratégia baseada em múltiplas *threads* consiste na alocação delas para o tratamento das URLs a serem coletadas. Quando várias estão conjuntamente atuando, enquanto algumas realizam a recuperação dos documentos de fontes como a Web, outras aproveitam para processar a informação recuperada, maximizando a utilização de recursos de hardware como ciclos de CPU e banda de Internet. Já uma única *thread* realizando assincronamente requisições às fontes de dados permite que diversos documentos sejam recuperados simultaneamente, criando um cenário onde é possível processar os documentos coletados paralelamente ao *download* deles e assim, como na estratégia empregando *threads*, maximizar os recursos de hardware.

A coleta paralela pode enriquecer a coleta distribuída sobremaneira. Cada processo distribuído executando em um determinado computador pode empregar tanto a estratégia

erguida sobre múltiplas *threads* quanto a sustentada por requisições assíncronas para melhorar o desempenho da solução final.

Ambas as estratégias (múltiplas *threads* e requisições assíncronas) apresentam resultados similares (HEYDON; NAJORK, 1999) e são empregadas em coletores conhecidos pela literatura. A versão inicial do coletor do coletor do Google utiliza conexões assíncronas, atentando para um máximo de 300 ocorrendo simultaneamente. Já o coletor Mercator emprega 100 *threads*, cada uma realizando requisições síncronas à Web e outras fontes. O coletor Internet Archive emprega um máximo de 64 conexões assíncronas simultâneas. O UbiCrawler utiliza 4 *threads* síncronas. O coletor Polytechnic lança um máximo de 1000 conexões assíncronas e o Thailand utiliza múltiplas *threads*, todas síncronas e totalizando 300. Os dados foram obtidos de Boswell (2003).

Normalização de URLs

URLs escritas de diferentes maneiras podem referenciar um mesmo documento. Por exemplo, as URLs http://www.google.com/, http://www.google.com e http://www.google.com/index.html, apesar de semelhantes, diferem sintaticamente pela segunda não terminar com uma barra e terceira explicitar o nome do arquivo HTML. Apesar da diferença, todas as URLs levam ao mesmo conteúdo na Web, o *site* do buscador da Google.

A coleta das três URLs, quando é necessário recuperar a página inicial do buscador da Google uma única vez, acarreta em perda de desempenho pelo coletor. Além do mais, coletores distribuídos e utilizando funções de particionamento baseadas na URL provavelmente coletarão cada URL em uma máquina diferente.

Para apurar esse problema, Gomes e Silva (2008) propõem a normalização das URLs durante o processo de coleta. Logo após identificar uma nova URL a ser coletada, esta deve prosseguir por um processo composto pelas etapas de adição de uma barra ao final da URL, remoção de âncoras, inclusão do prefixo ‘www’ para alguns sites e remoção de nomes de arquivos que podem ser omitidos.

Sobre a adição de uma barra ao final da URL, a especificação desse componente denota que se o caminho do arquivo não está presente, é necessário incorporar uma ‘/’ à URL

(BERNERS-LEE *et al.*, 1994). Esta regra previne que URLs como http://www.google.com e http://www.google.com/ originem dupla coleta de um mesmo documento.

Âncoras, como na URL http://www.google.com/index.html#ancora, são utilizadas para referenciar partes de um documento. Desse modo, a coleta de URLs que diferem somente quanto à âncora originará *downloads* repetidos.

Gomes e Silva (2008) observaram que a maioria dos documentos referenciados por URLs utilizando domínio de segundo nível (e.g. http://google.com/), estão também disponíveis quando formatadas com o prefixo 'www' (e.g. http://www.google.com/). A falta do prefixo 'www' em URLs de segundo nível responde por 51% do total de URLs duplicadas na Web. Sendo assim, a adição do prefixo reduz a quantidade de coletas duplicadas significativamente.

URLs semelhantes, que diferem somente pela presença ou não de um nome de arquivo conhecido, como em http://www.google.com/ e http://www.google.com/index.html, geralmente referenciam o mesmo conteúdo. A remoção do nome desses arquivos da URL reduz o número de coletas duplicadas em 36% (GOMES; SILVA, 2008).

Conteúdo Já Recuperado

Existem situações onde duas URLs irremediavelmente diferentes referenciam o mesmo conteúdo. Por exemplo, os endereços http://mail.google.com/, http://www.gmail.com/ e http://www.gmail.google.com/ orientam para o *webmail* da Google na Web. Nesse caso, mesmo que o processo de normalização de URLs, descrito na seção 2.2.6, seja aplicado às três URLs, a diferença sintática entre elas ainda será perceptível, levando o coletor ao processamento do mesmo conteúdo diversas vezes.

Diferentemente da proposta delineada pela normalização de URLs, que busca evitar múltiplas requisições HTTP para um mesmo documento, a verificação do conteúdo recuperado frustrará processamentos adicionais nele caso já tenha sido previamente coletado (HEYDON; NAJORK, 1999). Aqui, diversas requisições HTTP continuarão sendo realizadas para um mesmo documento a partir de distintas URL.

Após a identificação de URLs que referenciam o mesmo conteúdo, o coletor pode ainda normalizar uma URL em função de outras. Por exemplo, as URLs http://www.gmail.com/ e http://www.gmail.google.com/ seriam transformadas para

http://mail.google.com/, evitando múltiplas requisições ao mesmo conteúdo no futuro. Nota-se claramente a colaboração entre a atividade de verificação de conteúdo já coletado e a normalização de URLs para a melhora do desempenho do coletor.

Armazenar o conteúdo de todos os documentos para a comparação pode ser proibitivo, especialmente quando se ambiciona coletar toda a Web ou qualquer outra grande fonte de informação. Assim sendo, Heydon e Najork (1999) propõem o armazenamento do *checksum*[7] gerado a partir do conteúdo das URLs através de algoritmos como MD5 (*Message-Digest Algorithm 5*) e SHA (*Secure Hash Algorithm*). O *checksum* varia de alguns poucos *bits* a valores que podem chegar a 512 ou mais *bits* e pode ser armazenado tanto em memória principal, dentro de tabelas *hash*[8], quanto em disco, especialmente em estruturas como árvores B[9] e suas variações.

2.2.7 Tolerância à Falha

A resistência de um software na presença de falhas é um dos elementos que caracterizam um sistema de qualidade (KRITIKOPOULOS; SIDERI; STROGGILOS, 2004). Essa resistência é debatida no contexto de coletores por diferentes autores, como evidencia esta seção.

Para Kritikopoulos, Sideri e Stroggilos (2004), um coletor que lida com grandes quantidades de dados precisa manter a integridade da informação independente de quaisquer erros que porventura ocorram durante a coleta. Os autores também delineiam que, caso o processo de coleta pare repentinamente, ele deve estar habilitado a continuar a partir do momento em que se encontrava quando cessou, no pior dos casos com a perca de alguns poucos documentos que já haviam sido coletados, além de estar em conformidade com o padrão ACID[10] (atomicidade, consistência, isolamento e durabilidade) para o armazenamento dos dados coletados.

---

[7] http://en.wikipedia.org/wiki/Checksum

[8] http://en.wikipedia.org/wiki/Hash_table

[9] http://en.wikipedia.org/wiki/B-tree

[10] http://en.wikipedia.org/wiki/ACID

Hafri e Djeraba (2004) detalham aspectos que podem incorrer em erros de coleta e que devem ser mitigados. Coletores devem ao menos processar código HTML inválido, algo muito comum na Web, além de lidar com comportamentos não esperados gerados pelo servidor. O processo de coleta pode durar semanas ou meses, e é imperativo que o coletor lide com outros tipos de falha como processos de coleta parados, ou problemas na comunicação entre processos de coleta e também entre os processos de coleta e os servidores de dados. Finalmente, o sistema deve ser persistente, movendo periodicamente parte dos dados da memória principal para o disco, possibilitando assim o reinício do coletor caso necessário.

Boldi *et al*. apresentam a tolerância a falhas dado um contexto distribuído, com um coletor composto de vários processos atuando em computadores distintos:

> Assim que um agente morre abruptamente, o detector de falhas descobre que algo ruim aconteceu (e.g., utilizando *timeouts*). Graças à propriedade de função de particionamento, o fato de que diferentes agentes tem uma visão diferente do conjunto de agentes vivos não perturba o processo de coleta. Suponha, por exemplo, que *a* sabe que *b* está morto, enquanto *a'* não sabe. Por conta da contra variância, a única diferença entre *a* e *a'* na associação de *hosts* para agentes é o conjunto de *hosts* pertencentes a *b*. O agente *a* corretamente encaminha esses *hosts* para outros agentes, e o agente a' fará o mesmo assim que reconhecer que *b* está morto, o que irá acontecer, no pior caso, quando ele tentar enviar uma URL ao agente *b*. Neste ponto, *b* será marcado como morto, e o *host* enviado corretamente. Assim, *a* e *a'* nunca enviarão *hosts* para agentes diferentes. (BOLDI *et al*., 2004).

Nesse contexto, existe a expectativa de que o coletor tolere falhas esperadas nos processos, como no caso de uma manutenção programada no sistema de coleta, e também falhas inesperadas quando, por exemplo, ocorrem problemas com o hardware onde o coletor opera.

2.2.8 Múltiplos Protocolos e Tipos de Documentos

Diversos tipos de documentos estão espalhados por uma variedade de fontes de informação. Um coletor flexível deve entender essa pluralidade de fontes através dos protocolos necessários para a comunicação com cada uma, e também os tipos de documentos que podem estar contidos em tais fontes com o fim de entregar exatamente os dados que lhe é requisitado.

Um coletor que incorpora com precisão o entendimento de múltiplos protocolos e tipos de documentos é o Mercator (HEYDON; NAJORK, 1999). Ele provém módulos específicos tanto para o entendimento de múltiplos protocolos quanto para o tratamento de

diferentes tipos de documentos, suportando nativamente os protocolos HTTP, FTP e gopher, além dos tipos de documentos mais conhecidos como HTML, PDF e outros. Deste modo, o Mercator pode inspecionar a Web, repositórios de arquivos e outras fontes com facilidade, além de possibilitar a extensão através de novos protocolos e tipos de documentos dinamicamente.

A cooperação do coletor com novos protocolos é realizada geralmente pela utilização de bibliotecas de terceiros. Tecnologias como o Java[11] concretizam a especificação dos protocolos file, FTP, gopher, HTTP, HTTPS, news, NNTP e WAIS. Existe também uma gama de projetos, muitos *open source*, que provém o conhecimento necessário para a utilização de protocolos. Exemplos são o JCIFS[12] para o protocolo SMB, e o JavaSVN[13] para o protocolo utilizado na comunicação com o Subversion[14], um sistema de controle de versão empregado para o armazenamento e gestão das revisões de documentos.

O mesmo acontece com a manipulação de documentos, onde diferentes empresas e pessoas fornecem o software necessário para tal tarefa. Dentre os existentes, podemos destacar o Apache POI[15] para documentos do Microsoft Office, PDFBox[16] para documentos PDF, a JavaMail API[17] para leitura e composição de e-mails, além do HTML Parser[18] para arquivos HTML.

2.2.9 Armadilhas

Heydon e Najork (1999) definem uma armadilha como uma URL, ou conjunto de URLs, que pode causar a execução infinita do processo de coleta. Gomes e Silva (2008)

[11] http://java.sun.com/

[12] http://jcifs.samba.org/

[13] http://swik.net/javasvn

[14] http://subversion.tigris.org/

[15] http://poi.apache.org/

[16] http://pdfbox.apache.org/

[17] http://java.sun.com/products/javamail/

[18] http://htmlparser.sourceforge.net/

afirmam tal definição, considerando também como armadilhas qualquer situação que degrade o desempenho do coletor, mesmo que não origine coletas infinitas.

Armadilhas são criadas por um conjunto significativo de razões. Alguns *webmasters* produzem armadilhas na esperança que seus *sites* sejam melhor posicionados junto a mecanismos de busca (HEYDON; NAJORK, 1999), enquanto outros utilizam armadilhas para prevenir a extenuação de recursos de hardware pela atuação do coletor (GOMES; SILVA, 2008). Entretanto, armadilhas podem acarretar em problemas de usabilidade para humanos, comprometendo a navegabilidade do site caso sejam encontradas (GOMES; SILVA, 2008). Ainda, buscadores empregam técnicas para o reconhecimento de armadilhas, em muitos casos banindo *sites* Web que promovem o uso desse tipo de técnica (CHO; ROY, 2004).

Heydon e Najork (1999) descrevem as seguintes armadilhas: URL *aliases*, identificadores de seção imbuídos nas URLs e a geração de páginas Web dinâmicas. Gomes e Silva (2008) estendem essa lista com as armadilhas: mal uso de DNS *wildcards*, conteúdo de tamanho infinito e URLs que crescem. Essas armadilhas serão descritas até o final desta seção.

Duas URLs são *aliases* uma para a outra caso referenciem o mesmo conteúdo. Existem quatro casos onde isso pode ocorrer. O primeiro deles surge quando dois ou mais *hosts* apontam para o mesmo IP, como em gmail.com e mail.google.com. O segundo advém da omissão da porta nas URLs, como em http://www.google.com:8080/ e http://www.google.com/, com ambas referenciando o mesmo conteúdo. O terceiro é fruto de caminhos alternativos para um mesmo conteúdo, como em http://*host*/index.html e http://*host*/home.html. Finalmente, múltiplas cópias de um mesmo documento podem estar espalhadas por servidores distintos, incorrendo na coleta do mesmo conteúdo diversas vezes.

Sessões são geralmente utilizadas para a manutenção da comunicação persistente entre o cliente, geralmente um navegador Web, e também servidores de dados. Duas das técnicas comumente empregadas são a adição do identificador da sessão junto com a URL partindo do cliente para o servidor, e também a utilização de *cookies*. Apesar da técnica baseada em *cookies* não ser nociva ao coletor, a primeira pode fazê-lo coletar o mesmo conteúdo múltiplas vezes, já que o número de sessão adicionado ao final da URL pode não alterar o dado

recebido pelo cliente. Por exemplo, as URLs http://host.com/index.html;session=12 e http://host.com/index.html;session=34 provavelmente retornarão o mesmo conteúdo.

A geração de páginas dinâmicas é recorrente e implica, em muitos casos, a recuperação do mesmo conteúdo diversas vezes. Para a visualização desse problema, tem-se o exemplo da apresentação de um calendário por uma página Web que é atualizada toda a vez que a data é modificada. Nesse caso, é possível que cada dia gere uma página Web referenciando o mesmo conteúdo gerado pelo dia anterior, porém através de uma URL diferente, fazendo com que o coletor percorra todos os dias do calendário, ou seja, execute indefinidamente, sempre recuperando o mesmo conteúdo.

Outra armadilha refere-se à geração infinita de conteúdo para uma determinada requisição. Por exemplo, um programa mal intencionado pode suprir uma requisição HTTP indefinidamente, mantendo o coletor ocupado desnecessariamente. Existem também casos onde esse tipo de armadilha não foi deliberadamente concebida, como com rádios *on-line* que causam transmissões de dados infinitas.

Segundo Gomes e Silva, “uma armadilha pode ser criada através de um *link* simbólico de um diretório /spider para o diretório / e uma página /index.html que contém o *link* para o diretório /spider.” (GOMES; SILVA, 2008). Seguir esses *links* causará um infinito número de URLs (e.g., http://www.site.com/spider/spider/spider...), todos referenciando o mesmo conteúdo.

Um coletor robusto e flexível necessita conhecer as armadilhas descritas nesta seção e prover mecanismos para que sejam evitadas, não permitindo assim que o processo de coleta execute indefinidamente e também que recursos de hardware como banda de Internet sejam poupados.

2.3 Aplicações da Arquitetura

Uma arquitetura flexível para a coleta de dados deve estar apta a cooperar com uma extensa gama de cenários de coleta, com cada capturando necessidades específicas dos usuários por dados. A literatura acadêmica é abundante em exemplos que demonstram tais cenários, com alguns deles explorados nesta seção. São eles: buscadores, agregadores de notícias, coletores que interagem com comunidades sociais, coletores de informação sobre

proteínas, coletores que atuam na *hidden* Web, a reconstrução de *sites* Web, a busca por estruturas replicadas em fontes de dados e um estudo sobre a presença de *spywares* na Web.

Buscadores, comumente tratados por *search engines*, são ferramentas de extrema valia no atual cenário digital. Existe hoje uma enormidade de dados espalhados por uma grande quantidade de fontes, desde grandes como a Web até os discos de pequenos computadores pessoais que, cada dia mais, tem sua capacidade para a estocagem de dados ampliada. Nesse contexto, recuperar dados relevantes para atender uma necessidade específica do usuário é uma tarefa desafiadora. Lawrence e Giles (1998), Arasu *et al*. (2001) e Ntoulas, Cho e Olston (2004) enunciam os desafios tecnológicos para a construção de buscadores para a Web. Brin e Page (1998) anatomizam um buscador para Web e descrevem o que foi a versão inicial da ferramenta de busca do Google, descrevendo o módulo de coleta como "a aplicação mais frágil já que envolve a interação de centenas de milhares de servidores." (BRIN; PAGE, 1998). Em todos esses estudos, nota-se que um dos componentes que percorre perpendicularmente as soluções é o coletor, que proverá os dados necessários à materialização da solução de busca final.

Além dos buscadores, outro tipo de ferramenta que explora a coleta de dados são os agregadores de notícias. Agregador é um software que periodicamente lê um conjunto de fontes de notícias, em um dos vários formatos baseados em XML, e as exibe em uma única página. Diversos estudos foram propostos para incrementar a exibição das notícias. Henzinger *et al*. (2005) discute a descoberta de um subconjunto do total de notícias coletadas da Web relevante ao que está sendo correntemente transmitido por meios como a televisão. Beitzel *et al*. (2004) descrevem um conjunto de técnicas de filtragem para promover uma alta precisão durante a tarefa de apresentar notícias relacionadas a *queries* de busca. Gulli (2005) apresenta a anatomia de uma ferramenta de busca baseada em notícias. Sia, Cho e Cho (2007) apresentam algoritmos para o monitoramento de fontes geradoras de notícia com o intuito de que as notícias sejam recuperadas tão logo surjam. Tal qual acontece com os estudos acerca de buscadores, cada um dos trabalhos mencionados neste parágrafo necessita de um processo que colete notícias de diferentes fontes.

Dado a quantidade crescente de redes sociais dispostas na Web, estudos sobre como recuperar dados delas estão sendo visionados. Chau *et al*. (2007) apresentam um *framework* com coletores paralelos para a recuperação de dados contidos em redes sociais. Foi reportado pelo estudo a coleta de dados de aproximadamente 11 milhões de usuários, onde cerca de

66.000 deles tiveram seu perfil completamente inspecionado. Além da coleta de dados de redes sociais, também existem estudos sobre como utilizar os dados gerados por uma rede social para melhorar o processo de coleta, ou seja, como absorver pelo coletor os interesses de uma comunidade social e alimentá-la com dados que melhor atendam suas necessidades (PANT; BRADSHAW; MENCZER, 2003).

Estudos sobre proteínas geralmente produzem uma quantidade significativa de dados que são armazenados em bases variadas. Para a realização de muitos estudos acerca das proteínas, é necessário coletar e analisar parte dos dados armazenados por outros estudos. Por meio deste objetivo é que foi proposto o *Protein Information Crawler*, ou PIC, que automaticamente coleta dados sobre proteínas de uma ampla gama de fontes, sumarizando o que encontra em planilhas do Microsoft Excel ou tabelas HTML (MAYER, 2007). Segundo os autores, "o PIC acelera sensivelmente a busca por informação, ajudando na construção de bases de dados customizadas sobre proteínas que reduzem a investigação manual durante estudos protéicos extensivos" (MAYER, 2007).

Grande parte dos coletores hoje recupera páginas Web alcançáveis através de *hyperlinks*, ignorando outras que, para serem alcançadas, requerem o preenchimento de formulários (RAGHVAN; GARCIA-MOLINA, 2001). O conjunto de páginas alcançáveis através de formulários é chamado de *hidden* ou *deep* Web, e como não existem *links* para as páginas na *hidden* Web, elas não são descobertas pelos coletores e deixam de ser recuperadas. Alguns estudos endereçam esse problema. Ntoulas, Zerfos e Cho (2005) abordam a construção de um coletor que consegue, autonomamente, descobrir e coletar páginas dentro da *hidden* Web. Barbosa e Freire (2007) apresentam estratégias de coleta para a descoberta de pontos de entrada para a *hidden* Web.

McCown e Nelson (2006) oferecem um estudo sobre o desenvolvimento de um repositório para dados coletados a partir da Web, dados esses que serão utilizados posteriormente para a reconstrução de sites. Para tal, é realizada a coleta de páginas na Web a partir dos buscadores Internet Archive, Google, Yahoo! e MSN. Foram realizadas a reconstrução de 24 *sites* Web, com os resultados comparados às versões on-line dos *sites* coletados.

Muitos documentos aparecem replicados em fontes de informação como a Web. Algumas vezes, coleções inteiras deles são multiplicadas. Cho, Shivakumar e Garcia-Molina

(2000) descrevem como identificar tais coleções de documentos replicados onde o desafio é identificá-los a partir de massas de dados compostas de milhares de páginas Web e centenas de *gigabytes* de conteúdo textual. Os resultados foram alcançados utilizando uma base composta por 25 milhões de páginas Web coletadas em aproximadamente 150 *gigabytes* de dados.

Moshchuk *et al*. (2006) examinaram as ameaças envolvendo *spywares* dentro da Web. Para este fim, um coletor foi utilizado na recuperação de aproximadamente 18 milhões de URLs. Na massa de dados coletada, foram encontrados *spywares* em 13.4% dos 21.200 arquivos executáveis encontrados, e também identificados *scripts* maliciosos em 5.9% das páginas processadas.

Nota-se que, invariavelmente, todos os cenários apresentados dimensionam soluções de coleta específicas. A proposta do presente trabalho é a de compor uma arquitetura flexível de coleta, materializada em um software coletor de dados, que possa, ortogonalmente, atender cenários como os descritos nesta seção.

2.4 Trabalhos Relacionados

A arquitetura flexível da Figura 3 permite que seus itens sejam permutados livremente, tolerando uma enormidade de combinações e consequentemente um grande número de coletores implementados. Dos que fazem parte desse grande número, boa parte são segredos bem guardados de grandes companhias, com seus mecanismos não publicados por razões comerciais (HEYDON; NAJORK, 1999). Porém, existe uma parcela significativa de coletores descritos pela literatura acadêmica e alguns deles serão abordados nesta seção.

Heydon e Najork (1999) descrevem o Mercator, um coletor extensível e escalável escrito em Java. Extensível por sua arquitetura tolerar a configuração e troca de vários módulos, permitindo sua adaptação a uma grande quantidade de problemas de coleta, e escalável, pois o coletor foi desenhado para recuperar documentos de toda a Web. O Mercator permite ainda a coleta de documentos alcançáveis por outros protocolos que não o HTTP, aceitando a inspeção de outras fontes de dados que não a Web. O Mercator também permite a coleta de diversos tipos de arquivos. A Figura 9, retirada tal qual é apresentada por Heydon e Najork (1999), detalha a arquitetura do coletor.

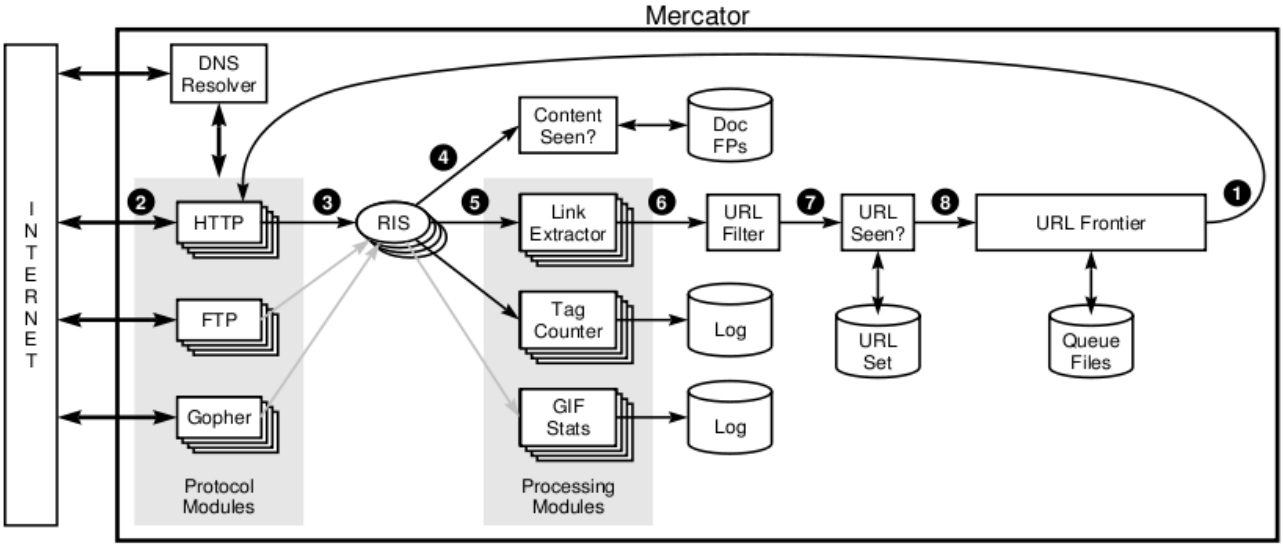


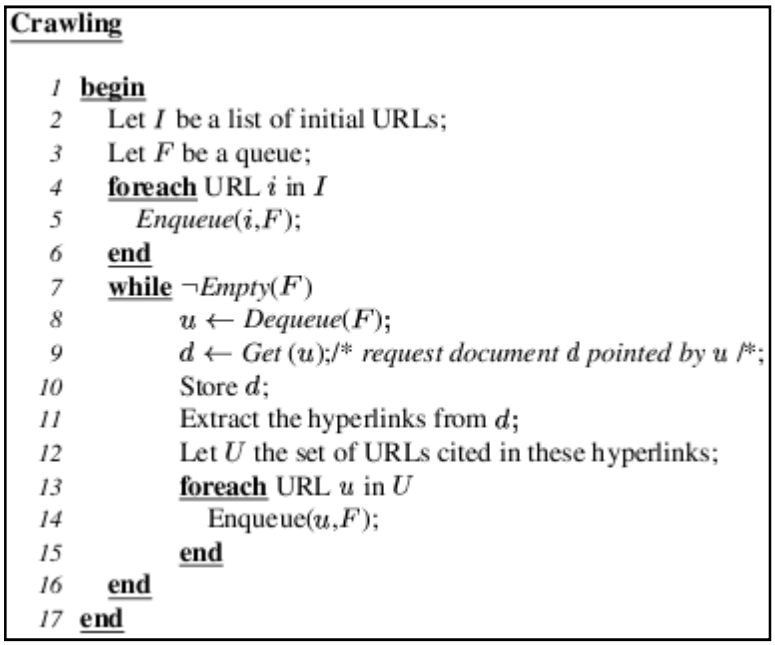
Figura 9. Arquitetura do coletor Mercator.

Fonte: Heydon e Najork (1999).

Primeiro, uma URL é removida do *frontier* 1. Baseado no protocolo definido pela URL, o módulo correto para o download do documento por ela referenciado é recuperado. O módulo é então invocado 2, e o retorno armazenado em uma estrutura de dados chamada *RewindInputStream* 3, uma abstração que permite com que o conteúdo do documento seja lido várias vezes sem a necessidade de múltiplas consultas à fonte de informação de onde este originou. Nesse momento o coletor verifica se o conteúdo do documento já foi obtido 4. Caso já tenha sido, o documento não é processado. Senão, baseado no MIME *type*[19], informação acerca do documento é recuperada, como referências para outros arquivos no caso de documentos HTML 5. Cada referência é convertida em uma URL, e o coletor identifica a necessidade de recuperá-las 6. Se a necessidade existir, o coletor verifica se essa já não foi coletada no passado 7 e, em caso negativo, a URL é adicionada no *frontier* para ser processada posteriormente 8.

Outro coletor é o CobWeb, apresentado por Altigran *et al*. (1999). Seu objetivo é o de recuperar grandes quantidades de documentos, observando os limites operacionais e éticos durante o processo de coleta. A Figura 10 detalha a arquitetura do coletor.

[19] http://en.wikipedia.org/wiki/MIME



```
Crawling

1  begin
2    Let I be a list of initial URLs;
3    Let F be a queue;
4    foreach URL i in I
5      Enqueue(i,F);
6    end
7    while ¬Empty(F)
8          u ← Dequeue(F);
9          d ← Get (u);/* request document d pointed by u /*;
10         Store d;
11         Extract the hyperlinks from d;
12         Let U the set of URLs cited in these hyperlinks;
13         foreach URL u in U
14           Enqueue(u,F);
15         end
16   end
17 end
```

Figura 10. Descrição da arquitetura do coletor CobWeb.

Fonte: Altigram *et al*. (1999).

Segundo os autores do CobWeb,

> Nesse algoritmo, as operações *Enqueue*, *Empty* e *Dequeue* são operações usuais sobre uma fila *F*, com as seguintes modificações. A operação *Enqueue* somente adiciona uma nova URL a ser removida caso ela já não esteja presente. A operação *Dequeue* somente marca a URL na frente da fila como "removida", ao invés de removê-la. Como consequência, *Empty* é verdadeiro somente quando todas as URLs na fila são marcadas como "removidas". Essa política para a manutenção da fila de URLs será chamada de *LWF* (*Longest Wait First*). (ALTIGRAN *et al*., 1999).

Shkapenyuk e Suel (2002) exibem o projeto e implementação de um coletor extremamente eficiente para a Web que executa sobre um *cluster* de computadores. O coletor é robusto na presença de falhas, além de adaptável a vários cenários de coleta como buscas em profundidade na Web por documentos, múltiplas coletas de um mesmo documento para manter a base coletada atualizada face às fontes de informação, coleta de documentos relevantes a um determinado tópico, além de caminhamentos randômicos pela Web e também a coleta de dados da chamada *hidden* Web.

Esses cenários foram balizadores durante o desenho da arquitetura do coletor, mostrada na Figura 11. Nela são dispostos dois componentes principais, chamados de *crawling application* e *crawling system*. O primeiro define quais URLs devem ser coletadas, enquanto o segundo realiza o download das URLs, devolvendo o conteúdo para o módulo

crawling application, onde será analisado e persistido (na imagem, os módulos *Storage System* e *other components*). O *crawling system* é ainda encarregado de políticas como a delineada pelos arquivos ROBOTS.txt (KOSTER, 1994), pelo controle da velocidade da coleta e pela resolução de *hosts* em IPs. Fica a cargo do *crawling application* a implementação de estratégias como a busca em profundidade e a busca por documentos de um determinado tópico.

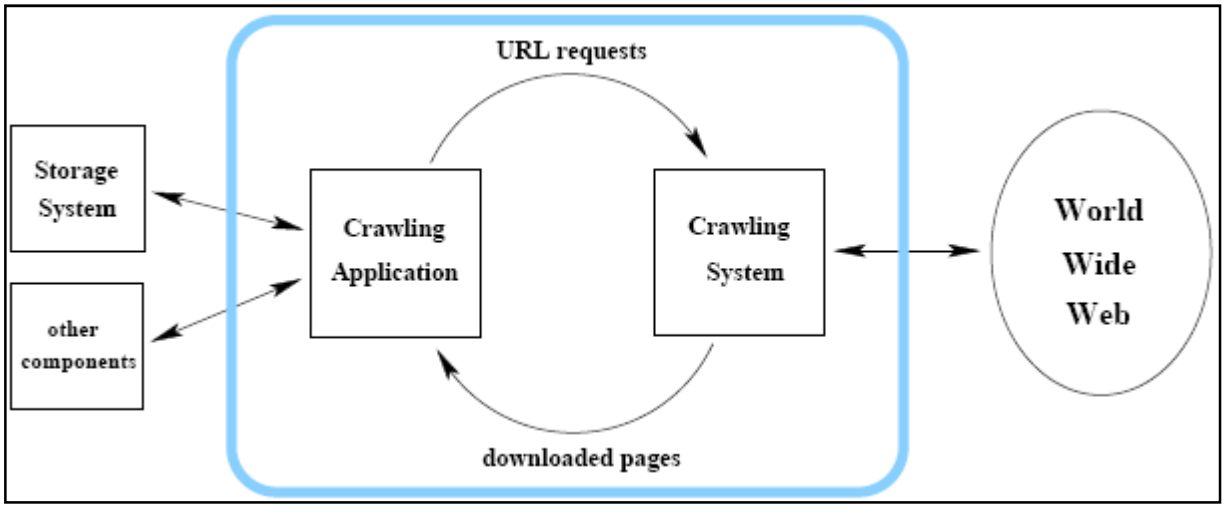


Figura 11. Arquitetura do coletor.
Fonte: Shkapenyuk e Suel (2002).

Boldi *et al*. (2004) demonstram o UbiCrawler, inicialmente nomeado Trovatore, um coletor escalável contendo uma arquitetura apoiada sobre componentes distribuídos. O objetivo principal dos autores foi de recuperar grandes conjuntos de dados da Web para estudar sua estrutura. Logo, é compreensível que as principais características sejam a independência de plataforma, a tolerância a falhas e um mecanismo eficiente para a cisão do domínio sendo inspecionado através de diversas tarefas de coleta.

A arquitetura do UbiCrawler é composta por diversos agentes, cada um responsável por determinado quinhão da Web. Eles operam através do algoritmo de busca em profundidade e são cuidadosos ao não buscarem dados de um mesmo *host* ao mesmo tempo. Durante o processo de coleta, as URLs descobertas por agentes que estejam fora do seu escopo de atuação, dividido *a priori*, são enviadas ao agente à direita, caminhando desta maneira por todos eles até encontrarem um que esteja apto a coletá-las. Já as URLs descobertas que pertencem ao domínio do agente que as encontrou são imediatamente coletadas.

Outros coletores que apresentam arquiteturas semelhantes às mencionadas são Dominos (HAFRI; DJERABA, 2004), o Viúva Negra (GOMES; SILVA, 2008), o descrito por Zhu *et al*. (2008) e também a versão inicial do Google Crawler (BRIN; PAGE, 1998).

3 PROJETO

A materialização do software coletor foi guiada pelo conjunto de artefatos descritos nesta seção. O projeto buscou, através da UML, capturar as características estruturais, comportamentais e de interação do software. Para a modelagem estrutural foi utilizado o diagrama de classe, e para capturar o comportamento do software os diagramas de caso de uso e sequência.

As seções que seguem abordam as funcionalidades necessárias a um coletor flexível através de uma listagem de requisitos (Seção 3.1), a modelagem das interações entre o usuário e o sistema através de casos de uso (Seção 3.2) e a modelagem dos componentes que formam o coletor por diagramas de classe e sequência (Seção 3.3). Os detalhes pertinentes ao desenvolvimento do projeto são descritos no Capítulo 4, junto com as técnicas para avaliação da solução.

3.1 Requisitos de Um Coletor Flexível

A literatura acadêmica aborda uma grande quantidade de requisitos de coleta. O conjunto de requisitos pertinente ao entalhe de uma arquitetura flexível através de um software coletor foi abordado e serviu de insumo para a diagramação das outras etapas do projeto. A listagem levou em consideração o anseio por um coletor que possa facilmente digerir diversas necessidades de coleta por parte dos usuários. Deste modo, foram identificados os requisitos dispostos na Figura 12.

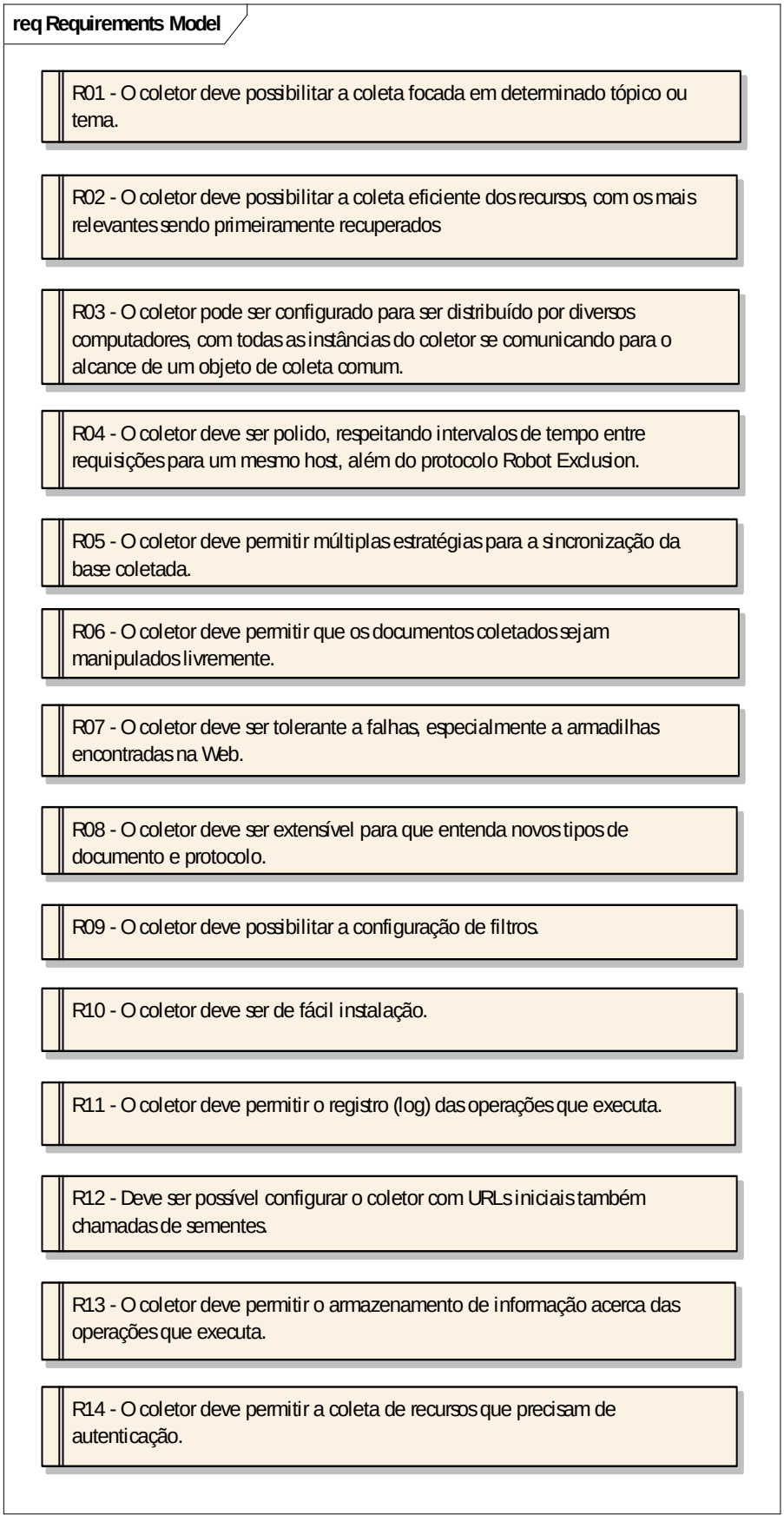


Figura 12. Listagem de requisitos do coletor.

Entende-se que a arquitetura de coleta que tomar essa lista de requisitos como suporte permitirá que variadas necessidades de coleta sejam referenciadas, tornando o coletor flexível. Será possível coletar tanto todos os dados contidos em uma fonte de informação quanto um conjunto deles relevante a um determinado tópico ou tema. Os documentos mais relevantes, de acordo com alguma medida informada pelo usuário, poderão ser coletados primeiramente em detrimento a outros documentos armazenados na fonte de informação e o coletor poderá ser executado tanto em uma máquina como em diversos computadores para que a capacidade de utilização de recursos de hardware como banda de Internet e armazenamento seja maximizada.

Ademais, o coletor respeitará boas práticas relativas ao contato com servidores de dados, permitirá múltiplas estratégias para a sincronização dos dados coletados junto às fontes de informação, será tolerante a falhas e permitirá o entendimento de novos tipos de documentos e protocolos dinamicamente. Será ainda possível configurar filtros para que somente os documentos necessários sejam coletados e também registrar as operações executadas pelo coletor.

Tabela 1. Mapeamento entre os requisitos e a fundamentação teórica

	2.1	**2.2.1**	**2.2.2**	**2.2.3**	**2.2.4**	**2.2.5**	**2.2.7**	**2.2.8**	
R01		■							
R02			■						
R03				■					
R04					■				
R05						■			
R06	■								
R07							■		
R08								■	
R09	■								
R12	■								

Os requisitos derivam do mapeamento exaustivo realizado sobre a literatura acadêmica na área de coletores e possuem uma forte relação com um ou mais tópicos abordados na fundamentação teórica (Capítulo 2). A Tabela 1 mapeia alguns requisitos (linhas) e a respectiva seção na fundamentação que o aborda (colunas). Os números dos requisitos constam na Figura 12.

3.2 Interações Entre o Usuário e o Coletor

Os clientes farão uso do coletor através de uma API (*Application Programming Interface*) contendo as funções e procedimentos necessários à operacionalização do software. Dependendo dos valores enviados e das operações invocadas através da API, o software coletor poderá assumir

comportamentos distintos. Essa amplitude de configurações é necessária para que múltiplos cenários de coleta, abstrações de necessidades de coletas por parte dos usuários, possam ser configurados, tornando o coletor e a arquitetura que esse materializa flexíveis.

O cliente nesta seção é modelado na UML por um ator, que é descrito como “alguém ou algo que interage com o sistema” (Eriksson *et al*., 2004). Cada interação é capturada por um caso de uso responsável pela descrição da troca de mensagens entre esses dois elementos. O ator envia uma mensagem ao sistema que responde ao ator com outra. Essa troca continua até que algum critério de parada seja atingido.

Os casos de uso são derivações dos requisitos. A listagem de casos de uso é guiada pelas funcionalidades que devem estar presentes no sistema, onde cada caso de uso busca detalhar a comunicação entre os usuários e o software. Assim, tomando a listagem de requisitos como insumo, os casos de uso são apresentados na Figura 13.

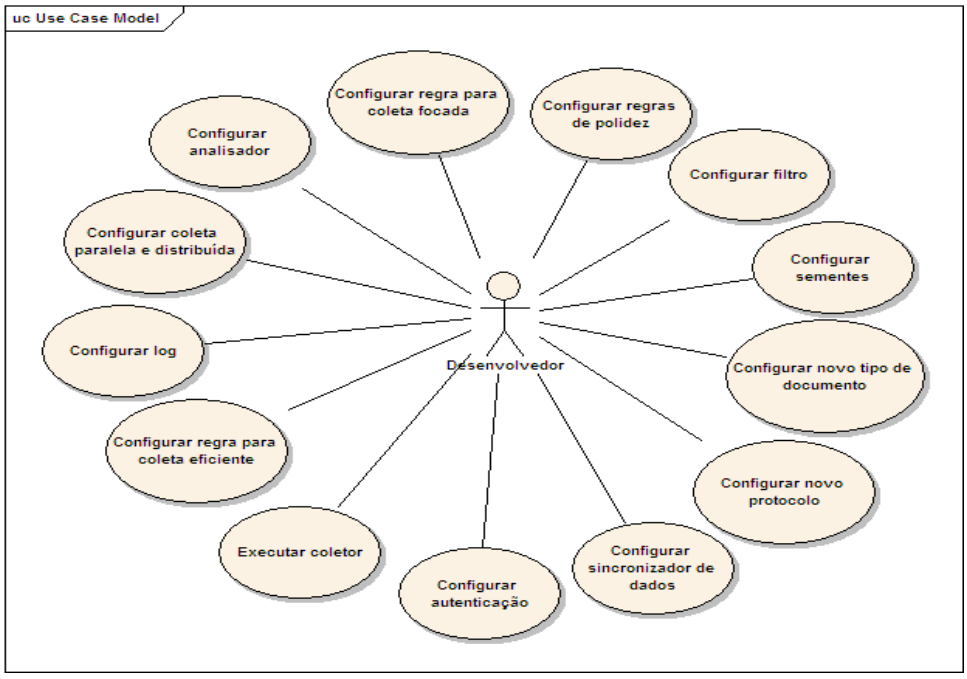


Figura 13. Ator e casos de uso.

O sistema é composto por onze casos de uso e um ator, que é:

- Desenvolvedor: é o responsável por configurar e executar o coletor.

Já os casos de uso são:

- Configurar regras para coleta focada: deve ser possível a seleção de um conjunto de opções junto ao coletor para que esse recupere somente documentos relevantes a um determinado tema;
- Configurar regras de polidez: o coletor necessita tratar polidamente os servidores que contata para não extenuá-los. Múltiplas requisições a um mesmo servidor na busca por dados, por exemplo, pode deixá-lo sem recursos para atender outras requisições. Ainda, o coletor precisa ser comedido com os recursos que utiliza para alcançar documentos, como banda de Internet e ciclos de CPU. Esse caso de uso detalha a interação entre o desenvolvedor e o coletor para que regras de polidez sejam configuradas;
- Configurar filtro: filtros são necessários para que somente documentos relevantes ao usuário sejam retornados. Por exemplo, através de filtros é possível selecionar somente documentos de determinado tipo ou provenientes de determinado *host*. Esse caso de uso mostra a interação necessária entre o desenvolvedor e coletor para que a funcionalidade seja configurada;
- Configurar sementes: sementes são URLs utilizadas pelo coletor como ponto de partida para o processo de coleta. Elas carregam informações relevantes, sendo elas o protocolo utilizado para a recuperação dos documentos, o local onde a coleta deve iniciar e por qual recurso. Por exemplo, a URL http://www.google.com/analytics/index.html informa que é necessário o entendimento do protocolo HTTP por parte do coletor, que a fonte de dados a ser inspecionada é o Google Analytics, uma ferramenta para a realização de métricas de páginas Web do Google, e que o processo de coleta deve iniciar pelo recurso index.html. É imprescindível que sejam configuradas sementes antes do processo de coleta ser iniciado, caso contrário o coletor não executará;
- Configurar novo tipo de documento: a quantidade de documentos passíveis de coleta é enorme e configurar todos os tipos dentro do coletor seria inviável. Deste modo, é necessário possibilitar ao desenvolvedor a configuração do coletor para que novos tipos de documentos possam ser entendidos. Esse caso de uso delineia a troca entre o desenvolvedor e coletor para que isso seja possível;
- Configurar novo protocolo: do mesmo modo que com os tipos de documento, os protocolos utilizados na recuperação da informação são muitos e configurar todos dentro do coletor é impraticável. Sendo assim, esse caso de uso define o conjunto de interações

entre o desenvolvedor e coletor para que novos protocolos sejam entendidos pelo software de coleta;

- Configurar sincronizador de dados: este caso de uso descreve os passos necessários para que seja configurado junto ao coletor algum algoritmo para sincronização dos dados coletados, como os vistos na Seção 2.2.5;
- Configurar autenticação: descreve os passos necessários para que o desenvolvedor configure o coletor com o objetivo de coletar documentos alcançáveis somente depois de algum processo de autenticação;
- Executar coletor: este caso de uso mostra os passos necessários à execução do coletor;
- Configurar regras para a coleta eficiente: descreve as interações necessárias para que seja configurado junto ao coletor algum algoritmo para a coleta eficiente, como os abordados na Seção 2.2.2;
- Configurar *log*: as operações executadas pelo coletor podem ser armazenadas para posterior consumo, seja para fins de manutenção como a identificação de falhas no software, ou mesmo para a geração de estatísticas sobre a coleta. Esse caso de uso mostra a interação necessária para que *logs* sejam gerados pelo coletor;
- Configurar coleta paralela e distribuída: o coletor poderá executar sobre um cluster de máquinas, com todas trabalhando objetivando a execução de um mesmo cenário de coleta. Este caso de uso elabora a interação entre o usuário e o coletor para que o coletor seja distribuído por diversas máquinas; e
- Configurar o analisador: O analisador é a interface pela qual os documentos coletados podem ser manipulados. A configuração do analisador junto ao coletor é capturada por esse caso de uso.

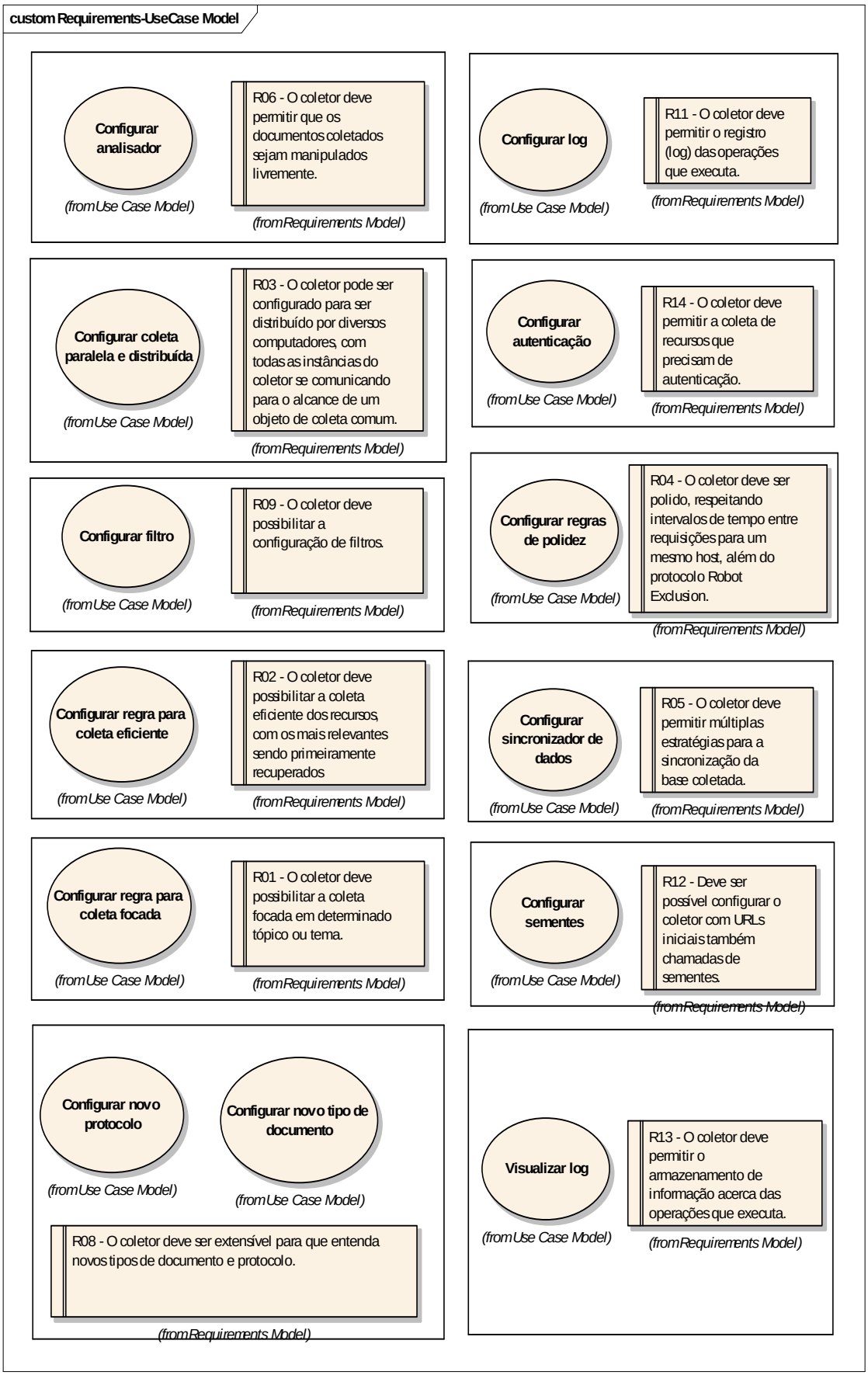


Figura 14. Rastreamento dos casos de uso.

Os casos de uso apresentados nesta seção derivam dos requisitos explicitados na Seção 3.1. É possível então mapear quais requisitos originaram que casos de uso como mostra a Figura 14

3.3 Componentes do Coletor

A interação entre o desenvolvedor e o sistema, mapeada nos casos de uso, é o material necessário para a descoberta e especificação dos componentes essenciais à codificação do coletor. Os componentes que perfazem o coletor são os blocos utilizados na composição da solução final, com a granularidade dependendo do foco necessário ao entendimento e codificação desses.

Aqui, a granularidade adotada foi dimensionada para receber cada uma das funcionalidades do coletor descritas na fundamentação teórica e abstraídas por requisitos e casos de uso. Por exemplo, a funcionalidade de filtro será contida em um único componente, assim como o código necessário para o desenvolvimento da coleta focada, eficiente ou mesmo a adição de sementes, todos temas contidos na fundamentação deste trabalho.

É possível, a partir da granularidade definida, conectar os blocos fundamentais à construção do coletor com os casos de uso do sistema, que por sua vez estão ligados aos requisitos e a toda fundamentação teórica explorada no trabalho. Assim, pode-se compreender de onde cada componente provém, como se dá a dinâmica de interação com o usuário, quais requisitos materializa e também que estratégias estão disponíveis pela literatura acadêmica para sua construção.

Além do foco no entendimento dos componentes, a granularidade apresentada também dá a cada componente o foco necessário ao seu desenvolvimento. Como cada componente mapeia um conceito explorado atomicamente pela literatura acadêmica, a codificação do conhecimento imbuído em artigos científicos será facilitada. Por exemplo, o componente responsável pela sincronização dos dados coletados pode ser rastreado até o caso de uso “Configurar sincronizador de dados” que deriva do requisito “O coletor deve permitir múltiplas estratégias para a sincronização da base de dados coletada” que provêm da Seção 2.2.5 sobre o conceito sincronização.

A codificação dos componentes em tal granularidade também permite a flexibilização da arquitetura, já que as estratégias para um coletor eficiente, apontadas pela fundamentação teórica,

poderão ser trocadas livremente para que um grande número de cenários de coleta possa ser atendido.

Durante o resto desta seção, cada um dos componentes será apresentado através de uma breve descrição, a estratégia de implementação a ser utilizada, um diagrama de classes para os aspectos estruturais e um diagrama de sequência para os comportamentais.

3.3.1 Analisador

O módulo analisador tem por objetivo permitir que os desenvolvedores possam dar um destino útil aos dados coletados. Por exemplo, através desse módulo é possível armazenar os documentos recuperados em banco de dados, indexar seu conteúdo para a realização de buscas textuais e gerar estatísticas sobre a informação coletada.

Como são muitos os fins para os dados coletados, a solução de coleta não pode fixar cenários de antemão. Nesse caso, o componente transfere para o desenvolvedor a responsabilidade por instanciar um módulo analisador com as regras que lhe convém para o destino aos dados coletados. Sendo assim, o coletor oferta não mais do que uma interface contendo apenas uma operação que será chamada a cada documento recuperado. Essa interface deve ser implementada pelo desenvolvedor e passada ao coletor.

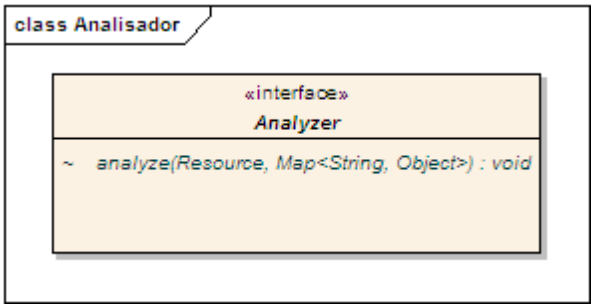


Figura 15. Interface do analisador.

A Figura 15 mostra o diagrama de classes para módulo analisador e a Figura 16 o diagrama de sequência com os passos para a criação de um analisador e configuração do coletor com este. É importante notar os argumentos da operação *analyze(Resource, Map<String, Object>)*. O primeiro argumento, o *Resource*, contém o documento coletado. Através desse objeto é possível recuperar

uma *stream* de *bytes* para o documento recuperado, bem como seu tamanho, tipo, *charset*[20], etc. O segundo argumento serve para que o desenvolvedor possa enviar informações acerca do documento coletado para outros componentes do coletor. Por exemplo, depois de inspecionar o conteúdo do documento, o desenvolvedor pode querer saber, a partir de outros módulos, se ele é importante. Para tal, o desenvolvedor pode adicionar uma propriedade no *Map*, o segundo argumento, da seguinte forma: *map.put("importante", Boolean.TRUE)*;. Não será explorado no projeto do coletor o mecanismo para a recuperação de tal informação em outros módulos.

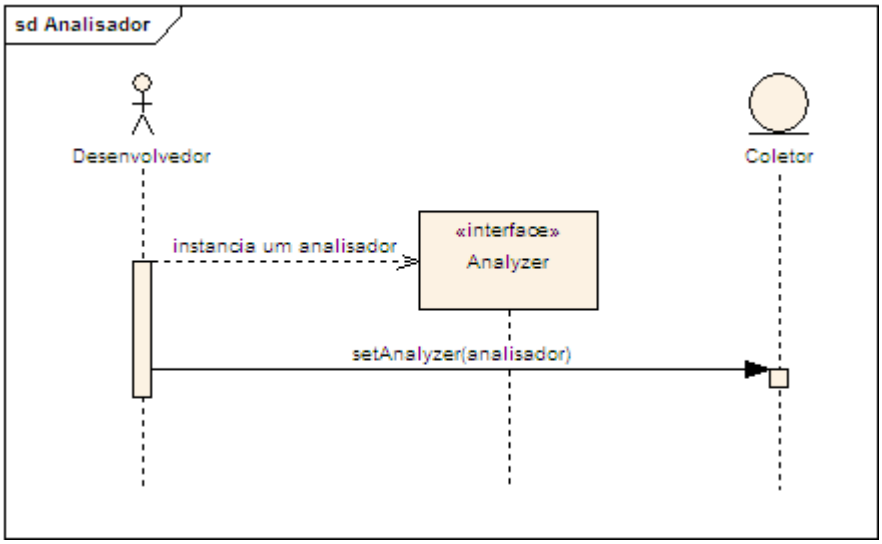


Figura 16. Configuração do coletor com um analisador.

O coletor será inicialmente configurado com uma versão nula do analisador através do padrão de projeto *Null Object* (HENNEY, 2003). O analisador, chamado *NullAnalyzer*, é uma implementação vazia da operação *void analyze(...)*. Assim, caso o desenvolvedor não configure analisador algum, os documentos coletados serão descartados. A Figura 17 mostra o código do *NullAnalyzer* escrito em Java.

```
public class NullAnalyzer implements Analyzer {

        public void analyze(final Resource resource,
                        final Map<String, Object> additionalInfo) {
        }
}
```

Figura 17. Configuração do coletor com um analisador.

[20] http://en.wikipedia.org/wiki/Character_encoding

Apesar de o coletor prover somente o analisador *NullAnalyzer*, a utilização do paradigma de desenvolvimento orientado a componentes para o projeto dessa parte do software permite que sejam criados e compartilhados analisadores para as tarefas comuns, como o armazenamento dos dados coletados em um banco de dados relacional ou indexação da informação recuperada. Assim, o compartilhamento de tais analisadores entre desenvolvedores é facilitado.

É necessário fazer a ligação entre esse componente e os casos de uso. Nesse caso, o componente deriva tanto do caso de uso "Configurar analisador" como do "Configurar novo tipo de documento". Assim, fica definido que o entendimento dos documentos coletados, que para o coletor é somente um fluxo de *bytes*, fica a cargo do desenvolvedor, que deve conceber o código necessário ao *parser* dos *bytes*.

3.3.2 Autenticador

O módulo de autenticação é necessário para que tanto documentos alcançados através de um *proxy* ou formulários de autenticação HTML sejam coletados. Um *proxy* é "um processo de gerenciamento de sessão com permissão para agir em nome de um usuário por um período limitado de tempo" (FOSTER *et al*., 1998). Já formulários de autenticação HTML geralmente compreendem um campo destinado para o nome do usuário e outro para a senha.

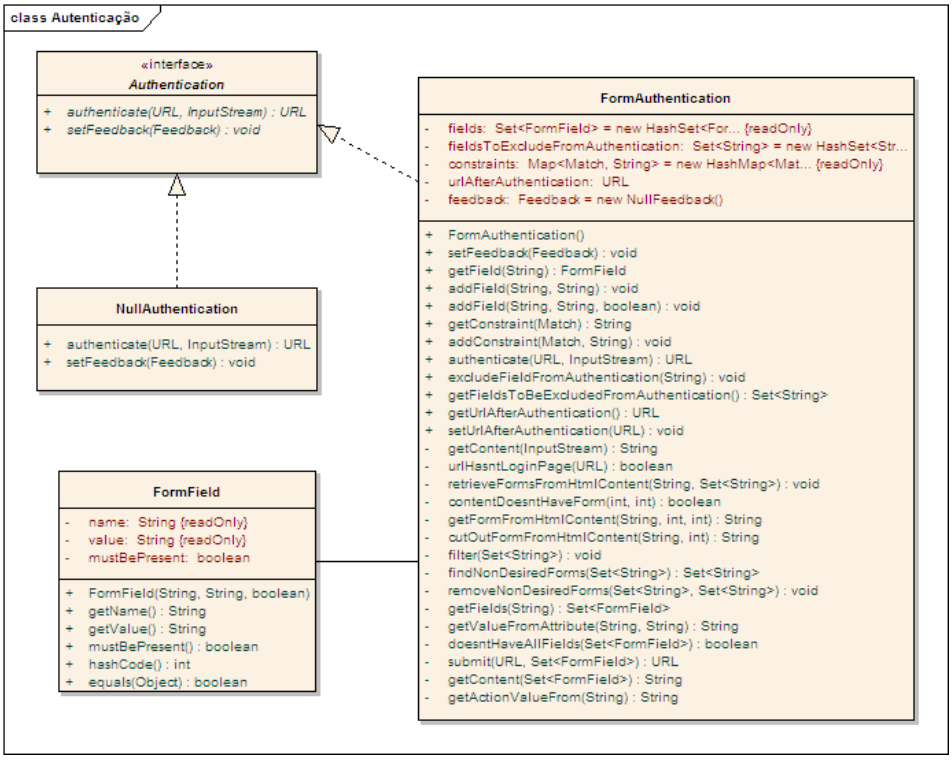


Figura 18. Diagrama de classes do autenticador.

A Figura 18 mostra o diagrama de classes para o módulo de autenticação. Nela é possível notar a interface que define um autenticador, bem como a implementação de um autenticador seguindo o padrão *Null Object*, o *NullAuthentication*, e também um autenticador para formulários HTML que requerem a digitação de um nome de usuário e de uma senha (*FormAuthentication*).

Se não for especificado pelo desenvolvedor, o coletor não utiliza qualquer mecanismo de autenticação, ignorando formulários HTML e não executando caso seja necessário passar as requisições por um *proxy*. Ou seja, o desenvolvedor precisa explicitamente configurar esse componente no coletor. O módulo de autenticação está atrelado ao caso de uso “Configurar autenticação”.

3.3.3 Feedback

Este módulo evidencia informações sobre a execução do coletor ao desenvolvedor. Isso se justifica, pois durante a coleta o sistema providencia *feedbacks* acerca de sua execução que podem ser capturados para uma variedade de fins.

Através do componente para a captura de *feedbacks* é possível registrar falhas que ocorreram no sistema, problemas durante a coleta de determinados documentos como *timeouts* e conexões encerradas pelo servidor, entre outras informações.

As informações providas são classificadas em três níveis: (i) *info*, contendo apenas informação acerca da coleta que pode ser relevante ao desenvolvedor; (ii) *warning*, com mensagens que devem ser analisadas com cuidado, pois podem intervir no bom comportamento do coletor; e (iii) *error*, apresentando informação sobre *bugs* que muito provavelmente estarão afetando negativamente a execução do coletor.

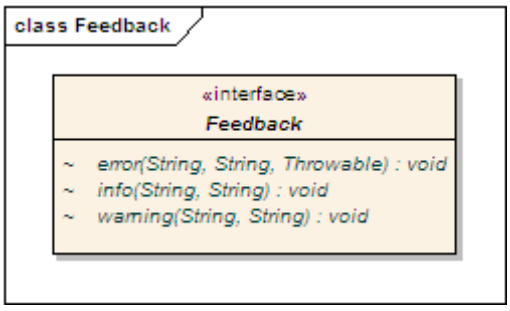


Figura 19. Interface do módulo *feedback*.

O módulo de *feedback* é definido por uma interface, como mostra a Figura 19. Ela provê operações que devem ser implementadas e passadas ao coletor. Toda a vez que um *feedback* é gerado pelo coletor, alguma dessas operações são chamadas e o código definido pelo desenvolvedor para elas é executado. O diagrama de sequência para a configuração do módulo de *feedback* junto ao coletor é mostrado na Figura 20.

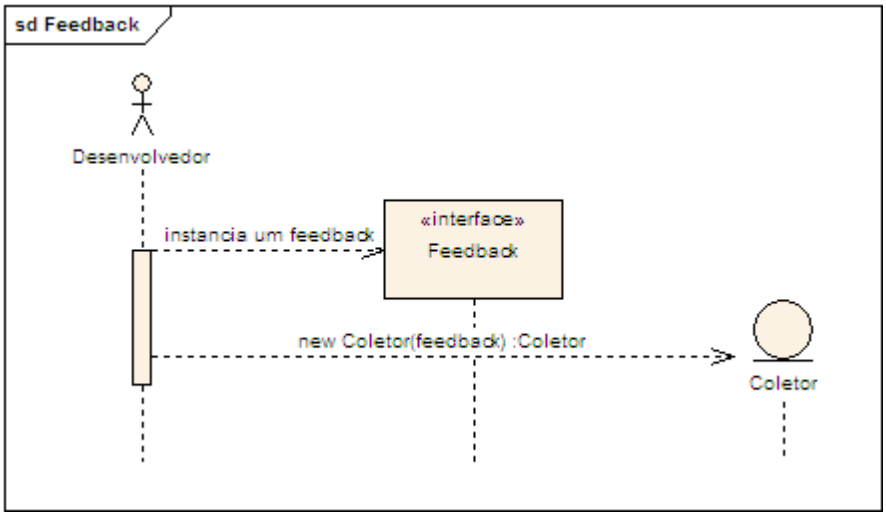


Figura 20. Configurador do coletor com o módulo *feedback*.

A implementação padrão é o *NullFeedback*, que deriva do padrão *Null Object* para que os *feedbacks* sejam descartados. O coletor também oferta o *ConsoleFeedback* para que os *feedbacks* sejam escritos no *console*. Como se pode notar, o módulo segue o paradigma de desenvolvimento baseado em componentes, podendo ser substituído livremente para atacar uma ampla gama de necessidades dos usuários. Um exemplo de implementação, para a classe *ConsoleFeeback*, pode ser visto na Figura 21.

```
public class ConsoleFeedback implements Feedback {

	public synchronized void error(final String url, final String message,
			final Throwable caught) {
		System.out.printf("ERROR MESSAGE: %s\n", message);
		if (caught != null) {
			caught.printStackTrace();
			System.out.println();
		}
	}

	public synchronized void info(final String url, final String message) {
		System.out.printf("INFO MESSAGE: %s\n", message);
	}

	public synchronized void warning(final String url, final String message) {
		System.out.printf("WARNING MESSAGE: %s\n", message);
	}
}
```

Figura 21. Interface do módulo *feedback*.

O módulo *feedback* ataca o caso de uso “O coletor deve permitir o registro (*log*) das operações que executa.”

3.3.4 Protocolos

Para que os documentos sejam recuperados, é necessário entender quais são os protocolos envolvidos na comunicação entre o coletor e o software servidor de dados. Por exemplo, documentos na Web são quase todos alcançáveis através do protocolo HTTP, ou seja, para que o coletor consiga recuperar documentos desse meio esse protocolo precisará ser compreendido.

Além do protocolo HTTP, uma infinidade de outros existem e cada um pode ser necessário à coleta de documentos relevantes. Assim, a arquitetura flexível precisa, além de suportar os protocolos mais comumente utilizados, também permitir ao desenvolvedor a configuração do coletor para que outros protocolos sejam adicionados dinamicamente.

Na arquitetura, o módulo responsável por lidar com os protocolos é chamado de *Fetcher*. Um *Fetcher* recebe uma URL no seu construtor. De acordo com o protocolo evidenciado pela URL, o módulo *Fetcher* correto é instanciado e provê, entre outros métodos, o *fetch()* e *getInnerUrls()*. O método *fetch()* retorna um *Resource* contendo o documento referenciado pela URL utilizada na construção do objeto *Fetcher*. Esse objeto *Resource* será retornado ao usuário através do analisador (Seção 3.3.1). Já o método *getInnerUrls()* retorna as URLs referenciadas pela URL utilizada na construção do *Fetcher*. Essas URLs serão filtradas (Seção 3.3.5) e depositadas no *frontier* (Seção 3.3.6).

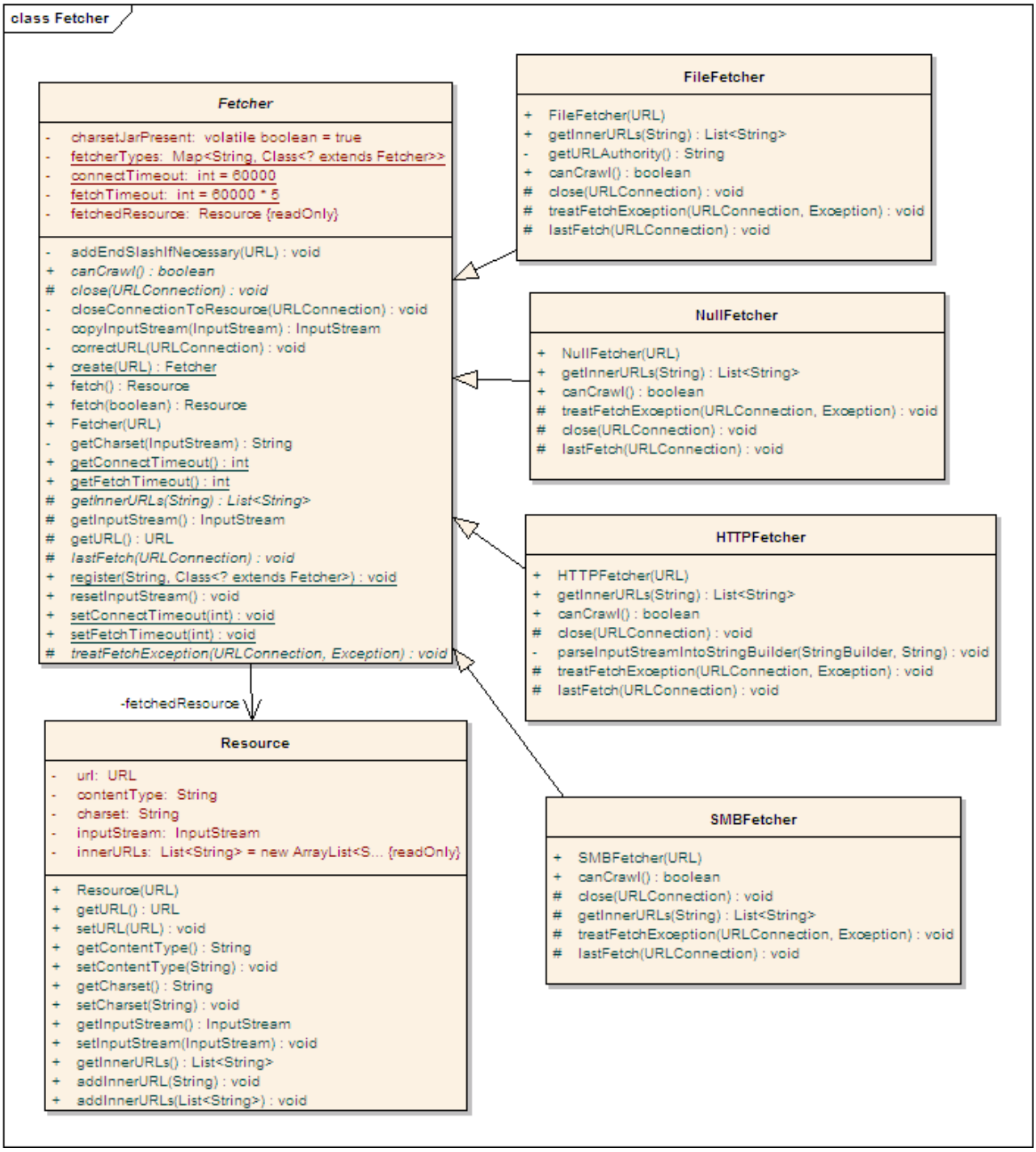


Figura 22. Estrutura de classes do módulo de protocolos.

A estrutura de classes desse módulo aparece na Figura 22. Nela, é possível notar que *Fetchers* para alguns protocolos comumente utilizados são apresentados, como o *HTTPFetcher*, *FileFetcher* e *SMBFetcher*, respectivamente para os protocolos HTTP, File e SMB.

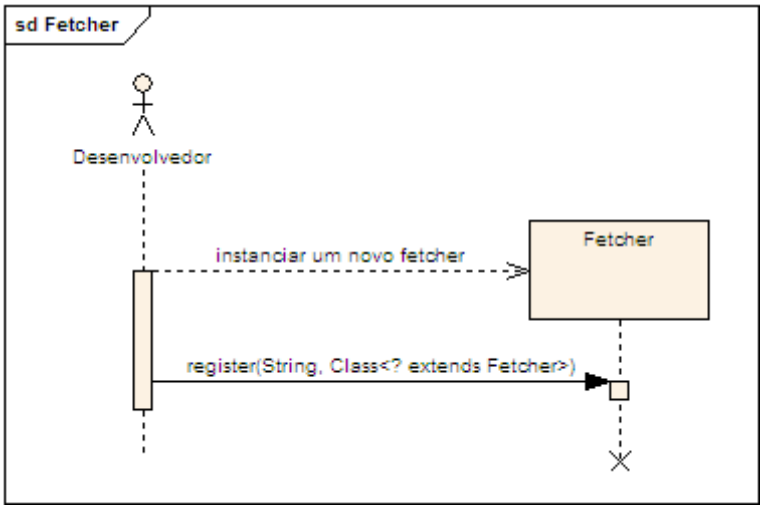


Figura 23. Configuração de um novo *fetcher*.

O desenvolvedor pode construir outros *fetchers* derivando da classe *Fetcher*. Assim, caso seja necessário coletar recursos alcançáveis através do protocolo SVN, por exemplo, é possível codificar um *fetcher* específico para tal objetivo e configurá-lo junto ao coletor, como mostra o diagrama de sequência da Figura 23. Na figura, primeiramente é necessário a codificação e instanciação de um novo *fetcher* para que esse possa ser posteriormente registrado junto à classe *Fetcher* através do método estático *register* que toma como argumentos o nome do protocolo e a classe *Fetcher* para atendê-lo. Esse módulo deriva do caso de uso “Configurar novo protocolo”.

3.3.5 Filtro

O componente de filtro é quem provê ao desenvolvedor o meio necessário para que somente os dados de interesse sejam coletados. É possível, através do filtro, eliminar certos tipos de documentos durante o processo de coleta como os que contêm determinada extensão (e.g. ppt, doc, rtf, jpg), nome e tamanho, ou ainda permitir que somente arquivos com determinadas características sejam coletados.

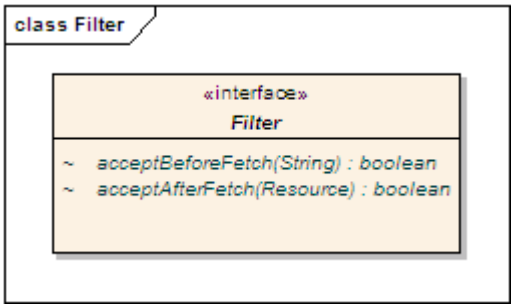


Figura 24. Interface do módulo de filtro.

A definição do filtro é feita através de uma interface, como mostra a Figura 24. São duas operações, sendo a primeira aplicada a URL dos documentos antes de requisições serem realizadas, e a segunda operação executada sobre o documento retornado pela URL. No primeiro caso, é possível utilizar somente a definição da URL para o filtro, já que o documento referenciado por esta ainda não foi recuperado. No segundo, uma gama informações pode ser utilizada, pois o documento recuperado possui o seu tamanho, conteúdo, *charset*, entre vários outros atributos.

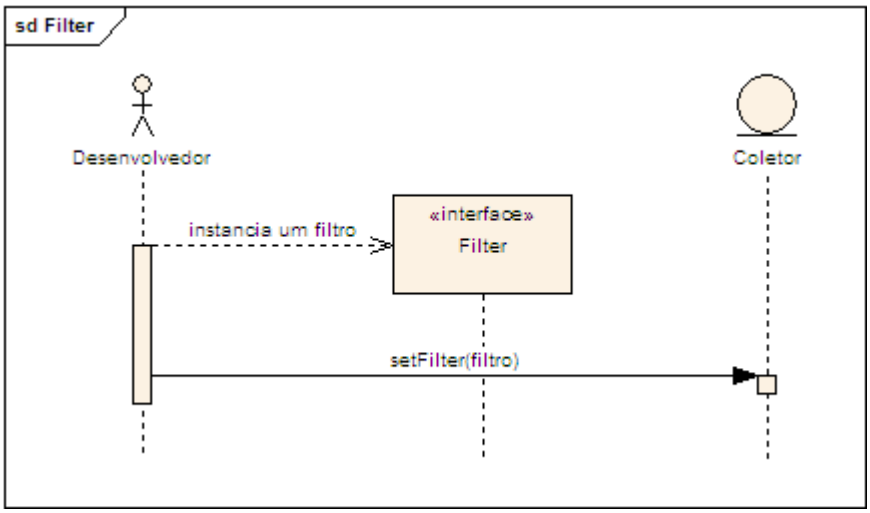


Figura 25. Interface do módulo de filtro.

Assim como acontece com outros componentes, o coletor utiliza por falta o *NullFilter*, uma implementação do padrão *Null Object* para esse módulo. O *NullFilter* não filtra documento algum, deixando todos serem livremente recuperados pelo coletor. Ainda, será ofertado mais um filtro, o *URLRegexFilter*, que verifica a URL dos documentos eliminando ou aceitando os mesmos de acordo com a avaliação de uma expressão regular. A Figura 25 exibe a sequência de passos necessários à configuração do coletor com um filtro. O módulo para a filtragem dos documentos está relacionado ao caso de uso “Configurar filtro”.

3.3.6 Frontier

O *frontier* é o módulo responsável por estocar as URLs a serem coletadas. De tempos em tempos, o *frontier* é inspecionado na busca por novas URLs que são removidas dessa estrutura, coletadas e devolvidas juntamente com as URLs referenciadas pelo documento recuperado. O *frontier* é inicialmente carregado com as sementes ou URLs iniciais.

Existem duas operações básicas em um *frontier*, sendo uma delas chamada de *enqueue* e a outra de *dequeue*. A primeira permite que URLs sejam dispostas dentro do *frontier*, enquanto a

segunda possibilita a retirada de URLs para serem coletadas. O diagrama da Figura 26 mostra e estrutura estática desse componente. Na figura é possível notar outras operações a serem utilizadas por outros componentes do coletor.

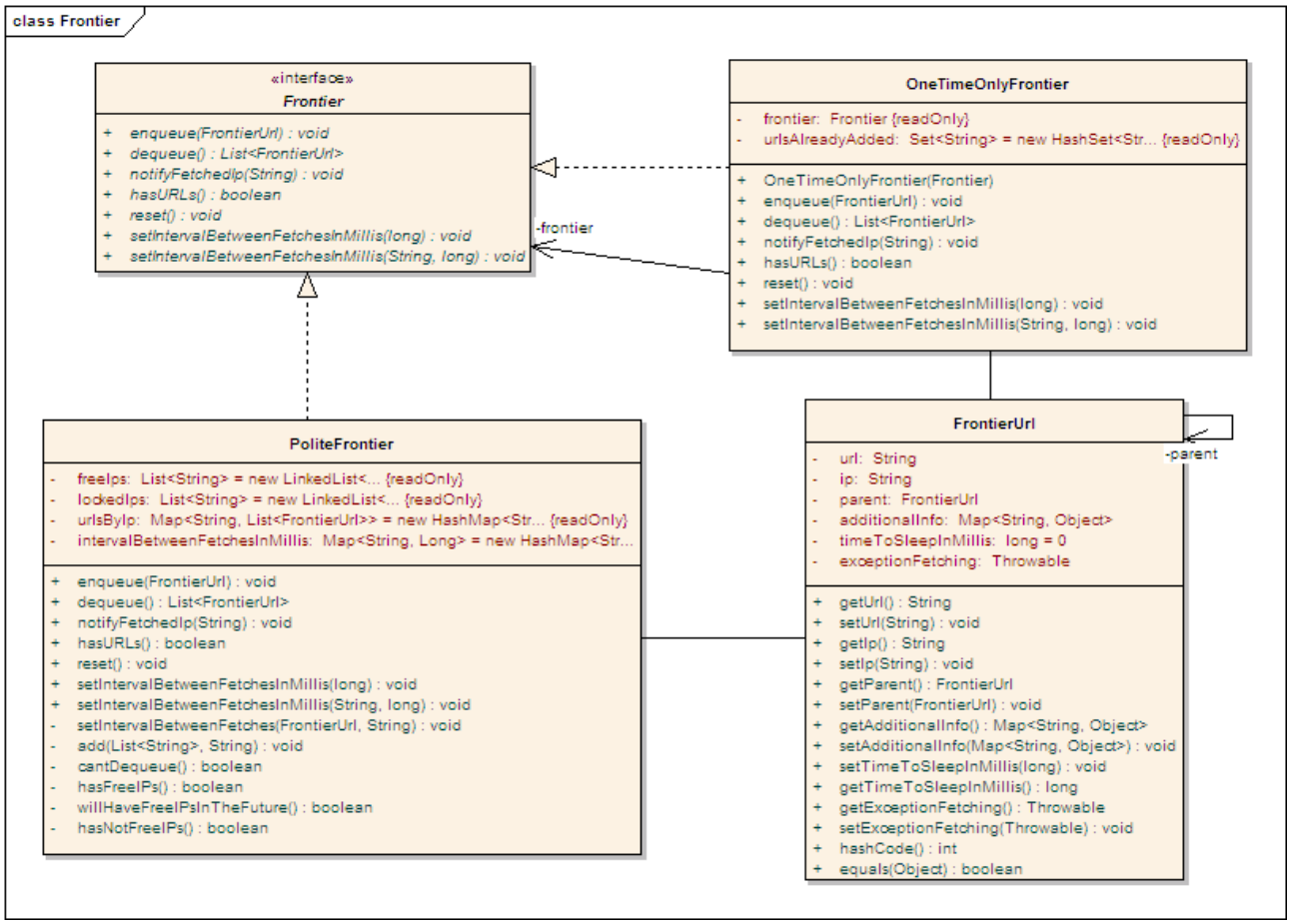


Figura 26. Diagrama de classes para o *frontier*.

Além das duas operações para adição e remoção de URLs dentro do *frontier*, o diagrama de classes detalha toda uma proposta para um *frontier* polido e que, opcionalmente, também permita que determinado recurso seja coletado uma única vez. A polidez é alcançada através da classe *PoliteFrontier* que, além de liberar uma URL por vez para um determinado *host* através do método *dequeue()* e somente quando a URL liberada anteriormente tiver sido coletada (evento notificado através do método *PoliteFrontier#notifyFetchedIp(String ip)*), também define um intervalo de tempo entre cada retirada de URL. Assim, requisições para um mesmo *host* serão realizadas sem afetar negativamente o computador servidor de dados. Note que é possível definir o intervalo entre a retirada de URLs de um mesmo *host* através do método *PoliteFrontier#setIntervalBetweenFetchesInMillis(long tempo)*.

Existe ainda a classe *OneTimeOnlyFrontier*, responsável por barrar URLs que já estiveram presentes dentro do *frontier* para que não sejam novamente coletadas. Essa classe decora outros *frontiers*, delegando a eles as funções de armazenamento e entrega de URLs. Ainda, é possível perceber que as URLs são encapsuladas através de objetos *FrontierUrl*. Esses objetos possuem mais informações como IP e URL pai da URL encapsulada que são valiosas para a construção de *frontiers* robustos.

Como o *frontier* é o módulo responsável pela entrega das URLs a serem coletadas, uma série de outras características necessárias a coletores flexíveis podem ser abstraídas através desse componente, com o desenvolvedor podendo implementar e configurar o seu próprio *frontier* no coletor como mostra a Figura 27.

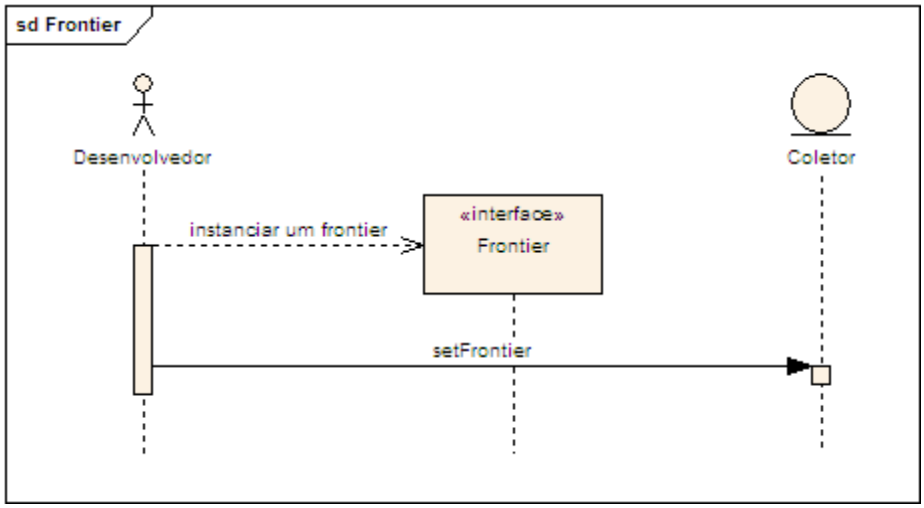


Figura 27. Configuração do *frontier* no coletor.

Além da polidez, referente ao caso de uso “Configurar regras de polidez”, outras características também podem ser abstraídas através do *frontier*. Uma delas é a coleta eficiente. O desenvolvedor pode imbuir no *frontier* heurísticas para que as URLs mais importantes sejam entregues primeiro para a coleta. Essa possibilidade endereça o caso de uso “Configurar regra para coleta eficiente”. Do mesmo modo, a coleta focada também pode ser implementada no coletor através do *frontier*, com esse entregando URLs para serem coletadas que tenham uma alta probabilidade de pertencerem a um tópico específico. O caso de uso “Configurar regra para coleta focada” é então atacado. Também o caso de uso “Configurar sincronizador de dados” é atendido pelo *frontier*, já que dentro desse componente é possível especificar quando entregar novas URLs ou quando enviar URLs já coletadas para serem atualizadas.

3.3.7 Executor

O módulo executor é o responsável pela orquestração de todos os outros componentes com o objetivo de coletar uma determinada URL. Ele é chamado inúmeras vezes até que URLs não mais existam dentro do *frontier* ou que algum critério de parada seja atingido.

O executor é definido através de uma interface, como mostra a Figura 28. É possível então implementar tal interface para que a execução de uma determinada URL ocorra de diversas maneiras. Por exemplo, é possível implementar um executor sobre um *pool* de *threads* para que cada URL seja recuperada em uma *thread* diferente, melhorando o desempenho da solução de coleta. Também é possível construir um executor onde URLs são recuperadas por máquinas distintas, assim distribuindo a coleta por diversos computadores.

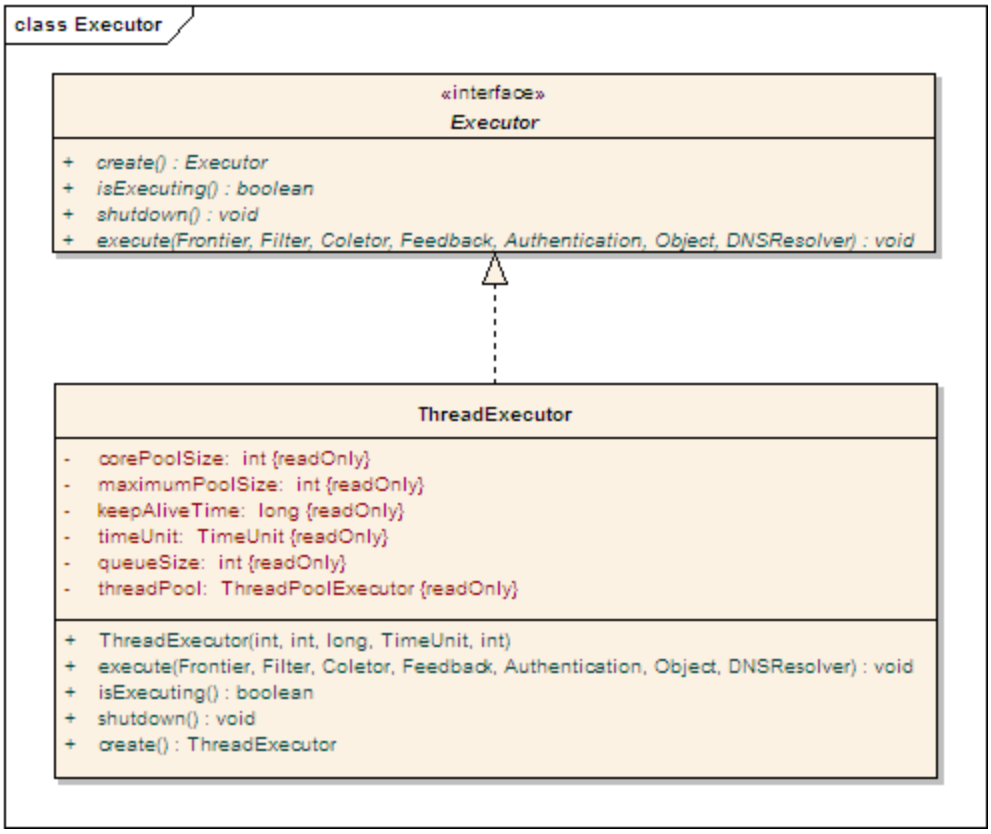


Figura 28. Diagrama de classes do módulo executor.

Na Figura 28, é possível notar a operação *execute*, que tem como parâmetros os outros módulos do coletor. Um exemplo de implementação para tal operação é a classe *ThreadExecutor*, também presente na figura, onde um executor baseado em *pool* de *threads* é utilizado.

É possível rastrear o módulo executor até o caso de uso “Configurar coleta paralela e distribuída”.

3.3.8 Integração dos Componentes

Os componentes foram apresentados de maneira isolada até aqui. Porém, para que o coletor funcione, esses precisam ser orquestrados com cada módulo cooperando com os outros para o objetivo comum de coletar dados.

Essa visão integrada dos módulos pode ser descrita através de um diagrama de sequência, como mostra a Figura 29. Nela, os módulos analisador, autenticador, executor, *feedback*, *fetcher*, filtro e o *frontier* cooperam para a realização da coleta documentos.

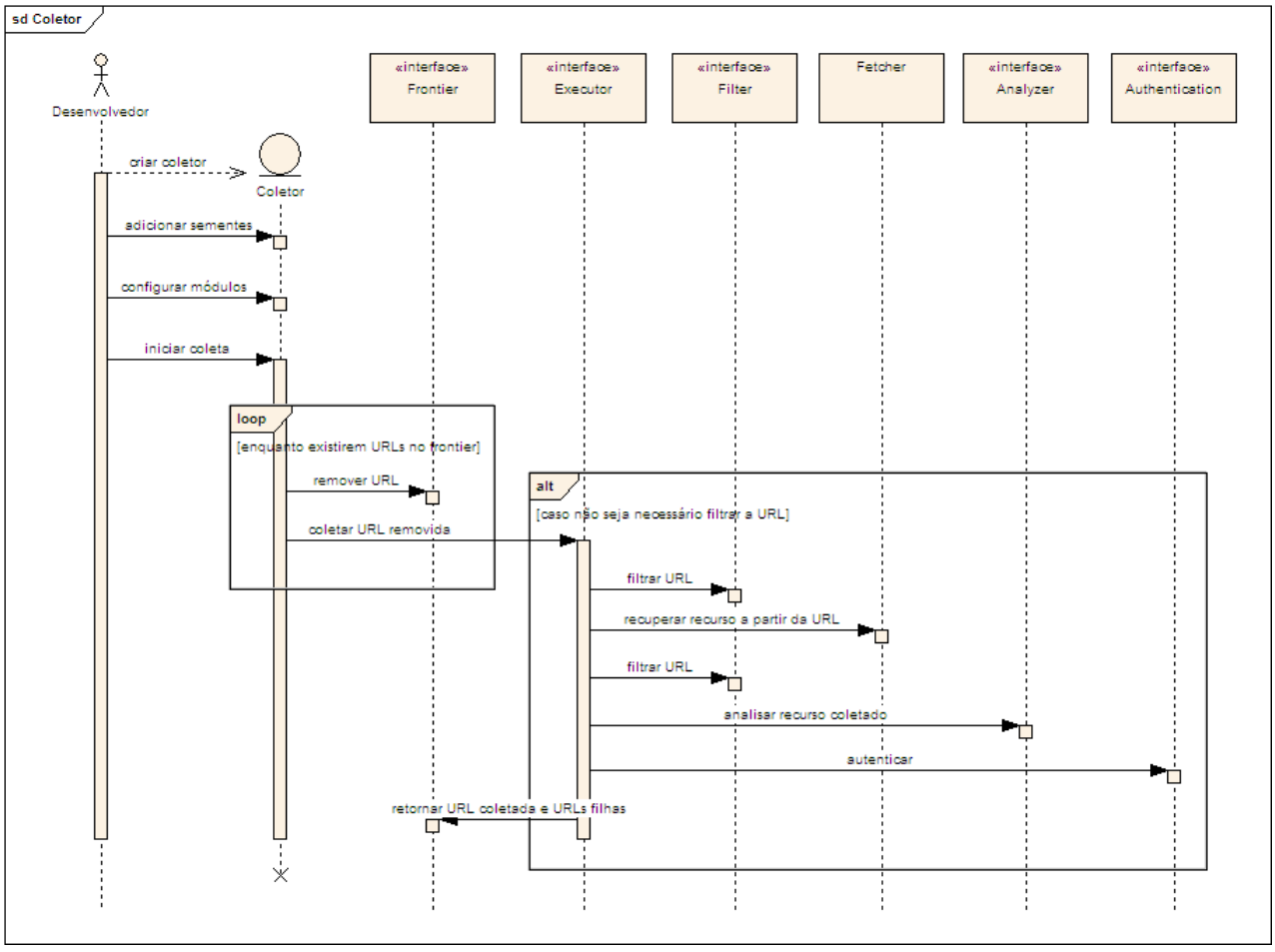


Figura 29. Diagrama de sequência com a execução do coletor.

Tem-se então o projeto da arquitetura flexível para a coleta de dados. No desenho dessa arquitetura, cada componente integrante foi detalhado e a ligação entre os componentes e os casos de uso explicitada, juntamente com a conexão dos casos de usos com os requisitos e desses com as

seções da fundamentação teórica. Assim, é possível traçar um paralelo entre as pesquisas recentes e relevantes à concepção de coletores flexíveis e a arquitetura proposta pelo trabalho.

4 DESENVOLVIMENTO

Para a construção do sistema foi utilizado o ambiente de desenvolvimento Eclipse, uma IDE (*Integrated Development Environment*) que, além de suportar a linguagem Java, é simples e interativa. A escolha da IDE a partir do universo de ambientes de desenvolvimento existentes se explica pela experiência do autor na utilização da ferramenta e também por esta apresentar todas as características técnicas necessárias à materialização do projeto em um sistema computacional. A linguagem de programação sobre a qual foi edificado o sistema computacional é o Java por sua oferta de classes necessárias ao desenvolvimento da arquitetura exposta na Seção 3.3.

Também foram utilizadas bibliotecas *open source* de terceiros para a construção de partes do coletor. Sem essa iniciativa, componentes do sistema construído que não integram o foco do trabalho tomariam um tempo demasiadamente longo para serem implementados, podendo comprometer a execução do cronograma.

Vale ressaltar que uma consulta foi inicialmente realizada à linguagem de programação escolhida pelos comportamentos delegados às bibliotecas de terceiros. Isso, pois se provido pelo Java, a necessidade de acoplar outros arquivos ao sistema, na forma de bibliotecas, seria eliminada. O autor também assume que, na falta de uma análise mais criteriosa, o código provido pela Sun (empresa que administra a evolução do Java) é mais eficiente, eficaz e menos propenso a defeitos. As seguintes bibliotecas foram utilizadas.

- icu4j-4_0.jar[21], necessária para a identificação do *charset* de um determinado documento coletado, ou seja, como um dado documento foi codificado para que possa ser manuseado corretamente pelo coletor;
- jcifis-1.3.1.jar[22], necessária para o entendimento do protocolo SMB, possibilitando o acesso a documentos compartilhados entre computadores de uma mesma LAN;
- junit-4.2.jar[23], necessária à realização dos testes de unidade.

[21] http://site.icu-project.org/

[22] http://jcifs.samba.org/

[23] http://www.junit.org/

Tomando como insumos a listagem de requisitos, os diagramas apresentados nas seções 3.1, 3.2, 3.3 e o ferramental tecnológico descrito nesta seção (IDE Eclipse, linguagem de programação Java e bibliotecas *open source*), tem-se o cenário utilizado na materialização da proposta desse trabalho, uma arquitetura flexível para a coleta de dados.

A definição dos componentes através da linguagem Java, pelo uso de classes abstratas e interfaces, pode ser encontrada no Apêndice A do trabalho. Os componentes lá dispostos são o analisador, o autenticador, o executor, o módulo para *feedback*, o módulo para o entendimento de diferentes protocolos (*fetcher*), o módulo para filtragem e o *frontier*.

4.1 Avaliação

Foram duas as etapas utilizadas na avaliação da arquitetura proposta. A primeira delas se deu pelo cumprimento de cenários pré-concebidos de coleta e a segunda pela comparação das características do coletor proposto contra uma lista de funcionalidades desejáveis a coletores flexíveis. Cada uma das etapas será explorada nas seções que seguem.

4.1.1 Cenários Pré-Concebidos de Coleta

A criação de cenários pré-concebidos de coleta objetiva a identificação de propriedades flexíveis da arquitetura proposta através do coletor que a implementa. O primeiro passo é a seleção dos cenários. Esses devem ser heterogêneos, requerendo que as características de coletores flexíveis apresentadas na fundamentação teórica sejam evidenciadas.

A seleção dos cenários se deu pela busca de artigos científicos que imbuíam em seu texto a necessidade por algum tipo de coleta de dados e também pela consulta a especialistas no domínio que enfrentam em seu dia-a-dia a necessidade por informação oriunda de diversas fontes. Os cenários escolhidos foram:

- **Coleta Livre de Dados a Partir da Web.** O cenário consistiu no uso do coletor para a recuperação de informação sobre documentos presentes na Web. A coleta é livre, pois partiu de algumas URLs manualmente definidas e seguiu suas ligações sem a utilização de qualquer tipo de filtro;
- **Coleta de Notícias.** Um primeiro processo varreu sites Web de jornais por *feeds*, documentos contendo um agregado de notícias formatadas no padrão XML para fácil

consumo. Uma listagem de *feeds* foi criada e fomentou um segundo processo que coletou periodicamente cada uma das notícias;

- **Coleta de Arquivos Compartilhados em LANs.** Foi realizada a identificação de todas as máquinas integrantes de uma mesma LAN e a consequente coleta de informação acerca dos documentos compartilhados entre elas;

A estrutura de cada cenário propõe cinco elementos. Primeiro, uma sucinta descrição do problema de coleta é apresentada. Logo após, é explanado como o coletor foi instrumentado para a correta execução do cenário de coleta. Na sequência, dados sobre a execução do cenário são apresentados. A seguir, a aderência do coletor ao cenário é debatida. As propriedades desejáveis a um coletor flexível atacadas pela execução do cenário são então demonstradas.

Coleta Livre de Dados a Partir da Web

Neste cenário, o coletor foi configurado para caminhar aleatoriamente pela Web e armazenar dados sobre os documentos encontrados para a posterior geração de estatísticas. O processo foi iniciado com um conjunto pré-selecionado de sementes e seguiu as ligações entre páginas HTML até a realização do critério de parada.

Uma das sementes utilizadas no processo de coleta foi a página da UOL (http://www.uol.com.br/). Essa semente foi escolhida, pois contém apontamentos para diversas outras páginas HTML, facilitando a dispersão do coletor por diferentes espaços da Web, um dos objetivos da coleta livre de dados. Outras sementes foram a página do Globo.com (http://www.globo.com/) e também do Yahoo! (http://www.yahoo.com.br/), ambas compartilhando da mesma propriedade mencionada para a primeira semente.

O critério de parada definido foi o número de dias executados pelo processo de coleta. O motivo foi a execução do coletor sobre uma LAN privada, utilizada por um instituto de pesquisa em suas atividades diárias. O administrador de rede definiu uma janela de tempo para que o processo de coleta, que extenua recursos como banda de Internet, pudesse ser executado sem afetar negativamente as operações do instituto.

Uma das alternativas para o prolongamento do tempo de execução do coletor foi limitar a quantidade de banda de Internet consumida por esse, porém a iniciativa foi prontamente descartada

visto que um dos objetivos do teste foi o de mensurar a velocidade da coleta, que seria demasiadamente afetada por tal decisão.

Pela grande quantidade de documentos manuseados, 556.271 ao todo, algumas estruturas do coletor, originalmente mantidas em memória principal, foram movidas para o disco. Por exemplo, o *frontier*, que armazena a lista de URLs por serem coletadas, foi transformado em uma tabela num banco de dados relacional já que a lista de URLs em memória principal certamente ultrapassaria os limites impostos pelo hardware. Os detalhes de configuração do coletor para esse cenário podem ser encontrados no Apêndice B.

A coleta foi executada a partir de um único computador com 2 Gb de memória RAM, CPU Intel Pentium 4 de 3.2 GHz contendo 2 núcleos, HD de 40Gb SATA com 7200 RPM, sistema de arquivos NTFS, placa de rede Broadcom NetXtreme 57xx Gigabit Controller e conexão, não dedicada, de 1 Mb com a Internet. Os dados gerados pela coleta foram armazenados no banco de dados HSQLDB.

Os 556.271 documentos foram coletados em três dias a uma velocidade de 2.04 documentos por segundo, perfazendo 23 Gb de dados. A Tabela 2 mostra a distribuição dos documentos de acordo com o seu tamanho. Nela é possível perceber a predominância de documentos HTML, seguidos por arquivos de imagem, scripts e outros.

Tabela 2. Quantidade de documentos coletados da Web por tipo.

Tipo de documento	**Documentos coletados**	**Tamanho médio em Kb**
HTML	411.162	52
Imagem	99.826	8
JavaScript	17.906	20
CSS	12.009	12
XML	6.038	34
PDF	1.282	310
Flash	1.154	52
Outros	6.894	676

Muitas das requisições para documentos incorreram em falhas. Para as 556.271 requisições que resultaram em documentos coletados, outras 54.535 não obtiveram sucesso, ou 8.92% do total de requisições HTTP. A Tabela 3 agrupa o número de requisições falhas pelo tipo de erro.

Tabela 3. Quantidade de requisições falhas por erro.

Tipo de erro	**Requisições falhas**	**% requisições falhas**	**% todas requisições**
Read timed out	12.943	23.73	2.11
Server returned HTTP response code	6.014	11.02	0.98
Connection reset	3.530	6.47	0.57
Premature EOF	2.015	3.69	0.32
Illegal char in URL	1.603	2.93	0.26
Connection timed out: connect	1.263	2.31	0.2
Connection refused: connect	275	0.5	0.04
Unable to find valid certification path…	264	0.48	0.04
Server redirected too many times	245	0.44	0.04
Outros	26.383	48.37	4.31

O erro mais comum foi o *read timed out*. Ele diz que uma determinada requisição superou o tempo de 5 minutos, limite definido pelo sistema de coleta, e foi abortada pelo coletor. Esse tempo pode ser configurado pelo usuário do sistema de coleta (ver método *setFetchTimeout(int)* da classe *Fetcher* na Figura 22). Outro erro recorrente ocorre quando o servidor encontra problemas na resposta ao coletor e retorna um código informando o motivo para o não atendimento da requisição. Exemplos de códigos são o 404, para documentos não encontrados dentro do servidor, e o 503, quando o servidor está demasiadamente ocupado para atender às requisições (FIELDING *et al*., 1999).

Mitigar os erros resultantes de requisições por parte do coletor é essencial para que seu desempenho melhore. Considerando a totalidade das requisições, em um cenário onde todas são executadas com sucesso, a velocidade de coleta seria calculada em 2.24 documentos por segundo, um aumento de 9.8%.

Apesar da execução do cenário demonstrar que é possível realizar a coleta de dados a partir da Web, a utilização da abordagem apresentada para a coleta de todos os documentos contidos nessa fonte é impraticável. Pela grande quantidade de documentos na Web, pelo grande número de atualizações desses e também pela adição constante de novos arquivos, aspectos como a distribuição do processo de coleta, a sincronização dos dados coletados com a fonte de informação e a tolerância à falha precisariam ser concebidos e configurados no coletor.

A proposta do cenário foi a de coletar livremente documentos da Web dada uma janela de tempo, armazenando dados acerca da execução do processo de coleta. Nesse contexto, a configuração do coletor se mostrou simples, sendo a construção de um novo *frontier*, que mantém

em disco e não em memória principal as URLs por serem coletadas, junto com a escrita de um analisador para a persistência dos dados sobre os documentos coletados em um banco de dados relacional, as únicas configurações relevantes realizadas no sistema de coleta.

Como todas as estruturas providas pelo coletor foram mantidas com exceção do *frontier*, e a nova construção desse módulo pode ser dinamicamente alocada ao coletor sem a necessidade de reescrita de código, o requisito flexibilidade se demonstrou verdadeiro neste cenário.

Coleta de Notícias

Este cenário propõe a coleta de notícias provenientes de jornais eletrônicos na Web. Nesse contexto, as notícias são usualmente disponibilizadas através de documentos XML conhecidos por *feeds*. *Feeds* são compostas por um determinado número de notícias onde cada uma tem suas partes devidamente marcadas para fácil consumo. Exemplos de partes de uma notícia são o título, conteúdo e autor.

Feeds frequentemente obedecem a um pequeno conjunto de padrões. Os mais comuns são conhecidos por Atom[24] e RSS[25], ambos utilizando XML como linguagem de marcação. Os padrões permitem que notícias de diversas fontes possam ser agregadas de maneira simples, eliminando a necessidade por uma solução de leitura de notícias para cada site Web. Esse foi um dos fatores essenciais para a execução do presente cenário já que notícias de um conjunto de fontes distintas foram capturadas.

A dinâmica da execução do cenário se deu em cinco etapas. Na primeira, sites Web foram indicados ao sistema de coleta. Logo após, o coletor varreu os endereços eletrônicos dos jornais apontados na procura por *feeds*. Assim que encontradas, cada *feed* teve sua URL armazenada em um banco de dados relacional. As URLs foram então inspecionadas periodicamente por notícias que foram persistidas em uma estrutura relacional.

Os jornais selecionados para a busca por *feeds* foram o Time (http://www.time.com/), The New York Times (http://www.nytimes.com/) e Folha de São Paulo (http://www.folha.uol.com.br/).

[24] http://en.wikipedia.org/wiki/Atom_(standard)

[25] http://en.wikipedia.org/wiki/RSS

A escolha foi fundamentada no fato de que todos os jornais citados possuem uma ampla base de leitores e também por esses liberarem notícias através de *feeds*.

O coletor foi inicialmente configurado com cada um dos sites Web dos jornais. Filtros foram adicionados ao coletor para que esse não extrapolasse os limites de cada jornal na Web. A configuração do coletor foi finalizada com um analisador para a inspeção de cada documento recuperado na busca por *feeds*. Cada *feed* encontrada foi devidamente armazenada em um banco de dados relacional para ter suas notícias consumidas por outro processo computacional.

A heurística aplicada na identificação das *feeds* inicia com a checagem da propriedade *content-type* dentro do cabeçalho das respostas HTTP enviadas pelos servidores de dados dos jornais selecionados. Essa propriedade informa o tipo de conteúdo retornado pelo servidor a partir de uma requisição a uma URL. Caso o valor da propriedade *content-type* contenha as palavras *atom*, *rss*, *feed* ou *xml*, o corpo da requisição HTTP é inspecionado pela biblioteca Rome[26], utilizada na manipulação de *feeds*. O objetivo da biblioteca é o de informar se o documento retornado é realmente uma *feed*. Caso seja, a URL da *feed* é armazenada para posterior consumo. A filtragem pelo valor do *content-type* evita que ciclos de CPU sejam gastos em vão através da inspeção de arquivos que de modo algum podem ser *feeds* como, por exemplo, documentos PDF ou do Microsoft Office.

Os sites Web dos jornais identificados são compostos por milhares de URLs. Nesse cenário, a inspeção de todo o corpus seria impraticável no tempo alocado para tal atividade. Destarte, uma janela de tempo de duas horas foi utilizada na identificação das *feeds*. A Tabela 4 mostra a quantidade de *feeds* encontradas para cada jornal, 145 no total.

Tabela 4. Quantidade de *feeds* por jornal.

Jornal	**Quantidade de *feeds***
Folha de São Paulo	27
Time	103
The New York Times	15
Total	145

[26] https://rome.dev.java.net/

A quantidade de *feeds* descobertas sobre o tempo em que o coletor executou aponta uma velocidade de 1.2 *feeds* encontradas a cada minuto. A baixa velocidade advém do fato de que os sites Web dos jornais inspecionados são compostos por milhares de documentos e uma pequena fração deles são *feeds*. Como somente a URL inicial de cada jornal foi emitida ao coletor (e.g. http://www.folha.uol.com.br/), esse precisou varrer grandes quantidades de documentos até encontrar os arquivos XML de interesse.

As *feeds* foram identificadas e armazenadas em um banco de dados relacional. Um segundo processo então foi configurado para inspecionar o conteúdo de cada uma das *feeds*, extrair as notícias dentro delas e armazená-las. Esse segundo processo não utilizou o coletor proposto. Ele foi composto por uma consulta SQL às URLs das *feeds* armazenadas e o *parser* delas utilizando a biblioteca Rome. Mais detalhes sobre a codificação deste cenário são encontrados no Apêndice C do trabalho.

O coletor proposto foi utilizado no cenário para a localização de fontes de notícias em sites Web de jornais. Ele conseguiu identificar um número considerável de *feeds* para cada jornal sem que suas estruturas internas precisassem ser reescritas. Somente filtros foram configurados, sementes adicionadas relativas aos sites Web de jornais, e um analisador para a identificação e armazenamento de *feeds* dinamicamente plugado ao coletor, demonstrando a estrutura flexível da arquitetura implementada por esse.

Coleta de Arquivos Compartilhados em LANs

Notebooks, PCs e impressoras são comumente agregados no que é chamado de LAN, uma rede local para dispositivos desse tipo. LANs são geralmente encontradas em organizações onde existe a necessidade de comunicação entre sistemas computacionais fisicamente separados para que objetivos institucionais sejam cumpridos.

A comunicação entre os diversos dispositivos integrantes de uma LAN, geralmente computadores, é realizada, em grande parte, pelo compartilhamento de arquivos entre as diversas máquinas da rede. Desse modo, é possível que um determinado documento seja lido ou escrito em outro computador.

Em grandes instituições, o número de documentos compartilhados pode atingir milhões. Nesse contexto, a localização do documento correto para o atendimento de uma necessidade qualquer pode se configurar uma tarefa árdua.

Existem hoje ferramentas especializadas nesse tipo de necessidade. Exemplos são o Copernic[27] e o Google Search Appliance[28]. Tais ferramentas varrem os computadores integrantes de uma LAN por compartilhamentos que possam fornecer documentos que serão indexados e se tornarão passíveis de busca. O presente cenário busca emular o processo de coleta de documentos de uma LAN tal qual é executado por essas ferramentas.

Inicialmente, foi identifica a LAN a ser utilizada no processo de coleta. Foi escolhida a LAN de uma instituição de pesquisa por essa conter um número razoável de computadores (72 ao todo), e também pelo fato do autor possuir acesso a ela.

Após a definição da LAN a ser utilizada, todos os seus computadores foram mapeados. Para isso, o protocolo ICMP foi utilizado na identificação de cada um dos IPs possíveis dentro da faixa de IPs utilizada pela organização. Os IPs válidos foram convertidos em URLs (e.g. smb://192.168.1.25/) e enviados ao coletor como sementes.

Tabela 5. Quantidade de documentos coletados da LAN por tipo.

Tipo de documento	**Documentos coletados**
Out	9.499
svn-base	5.808
Otf	4.746
HTML	3.753
Png	3.245
Jar	2.899
Jpg	2.032
XML	1.975
Java	1.432
As	1.356
Str	1.187
Gif	1.139
Class	1.111
Swf	889
Txt	718
Outros	12857
Total	54646

[27] http://www.copernic.com/

[28] http://www.google.com/enterprise/search/gsa.html

O coletor foi então acionado dentro de uma janela de tempo definida pelo administrador da LAN (2.5 horas) para que o processo de coleta não impactasse negativamente outros serviços. Os diretórios compartilhados foram inspecionados e os dados sobre os documentos encontrados armazenados em um banco de dados relacional. A Tabela 5 mostra a quantidade de documentos coletados por tipo. Os detalhes acerca da configuração do coletor para este cenário estão no Apêndice D.

O coletor de dados se mostrou flexível no atendimento do cenário de coleta em LANs. Não foi necessária a reescrita de nenhum de seus componentes, somente a configuração do sistema de coleta com algumas sementes e a definição de um analisador para persistir em uma estrutura relacional os dados sobre os documentos recuperados.

4.1.2 Características do Coletor

As características desejáveis a um coletor flexível de dados foram colhidas de Girardi, Ricca e Tonella (2006). Elas, junto com os cenários de coleta, suportam a identificação do quão flexível é a arquitetura para coleta proposta pelo presente trabalho. Idealmente, todas as características desejáveis devem estar presentes no coletor de dados construído sobre a arquitetura proposta.

Duas características foram suprimidas. São elas a realização da coleta de todos os *links* dentro de determinada página Web e a recuperação de páginas presentes em *hosts* diferentes dos indicados pelo usuário. O motivo é que essas duas características são essenciais a qualquer proposta de coletor flexível, sendo a discussão sobre o suporte delas irrelevante ao trabalho. As demais características são listadas abaixo.

- Filtro de diretório, onde o coletor recupera documentos se um nome de diretório aparece na URL;
- Filtro de profundidade, com o coletor recuperando todos os documentos até uma determinada profundidade a partir do documento raiz;
- Filtro de tipo, onde o coletor recupera somente documentos de determinado tipo, como imagens, ou arquivos PDF;
- Filtro de número de documentos, com o coletor recuperando os primeiros *N* documentos definidos pelo usuário durante o processo de coleta;

- Filtro de tamanho, com o coletor recuperando documentos que sejam menores que um determinado valor em *kilobytes*;
- Filtro de número máximo de *kilobytes*, com documentos maiores que um determinado limite sendo eliminados;
- Filtro de URL parcial, onde o coletor recupera somente documentos apontados por URLs que contenham determinado conjunto de caracteres.
- Facilidade de instalação e configuração, ou seja, se esse procedimento é extremamente simples ou não;
- Suporte de alguma interface gráfica;
- Execução do coletor em modo *batch*;
- Realização do *log* das operações que executa;
- Obediência à política definida pelos arquivos para a exclusão de coletores, também conhecidos por *Robot Exclusion Files*;
- Configuração de *proxy*;
- Utilização de múltiplas *threads* ou requisições assíncronas;
- Coleta distribuída por diversos computadores;
- Sumarização dos dados coletados; e
- Apresentação gráfica dos documentos coletados e suas interconexões.

A Tabela 6 sumariza todas as características, marcando aquelas suportadas em sua plenitude pelo coletor, quais são parcialmente suportadas e as que não são atendidas pela solução. Características suportadas na plenitude são as atendidas nativamente pelo coletor construído. As suportadas parcialmente não são encontradas no coletor proposto, porém podem ser dinamicamente adicionadas a esse. Já as características não atendidas são aquelas que não podem ser incorporadas ao coletor sem a realização de revisões na arquitetura.

Tabela 6. Características desejáveis a um coletor de dados flexível.

Número	**Característica**	**Suportada**	**Explicação**
1	Filtro de diretório	Sim	O coletor oferta o filtro *URLRegexFilter* para esse propósito (Seção 3.3.5).
2	Filtro de profundidade	Parcialmente	A característica pode ser incorporada ao coletor pela implementação e configuração de um novo filtro (Seção 3.3.5).
3	Filtro de tipo	Sim	Ver explicação da característica 1.
4	Filtro de número de documentos	Parcialmente	Ver explicação da característica 2.
5	Filtro de tamanho	Parcialmente	Ver explicação da característica 2.
6	Filtro de número máximo de *kilobytes*	Parcialmente	Ver explicação da característica 2.
7	Filtro de URL parcial	Sim	Ver explicação da característica 1.
8	Facilidade de instalação e configuração	Sim	O coletor construído foi empacotado como um arquivo *jar* que pode ser facilmente adicionado a qualquer aplicativo Java.
9	Execução em modo *batch*	Sim	É o modo de operação padrão do coletor. Outros podem ser configurados através do módulo *frontier* (Seção 3.3.6).
10	*Log*	Sim	O módulo Feedback (Seção 3.3.3) foi especialmente construído para tal propósito.
11	*Robots Exclusion Files*	Parcialmente	A característica não está imbuída no coletor, porém pode ser adicionada através de uma nova implementação do módulo executor (Seção 3.3.7).
12	*Proxy*	Parcialmente	Ver explicação da característica 11.
13	Múltiplas *threads* ou requisições assíncronas	Sim	O módulo executor (Seção 3.3.7) é o responsável por essa característica. A implementação padrão desse módulo, que pode ser vista na Figura 28, implementa tal característica.
14	Coleta distribuída	Parcialmente	Ver explicação da característica 11.
15	Sumarização dos dados coletados	Parcialmente	Sumários sobre os dados coletados podem ser realizados por uma nova implementação do módulo analisador (Seção 3.3.1).
16	Apresentação gráfica das interconexões entre os documentos	Parcialmente	Do mesmo modo que os sumários, qualquer apresentação gráfica dos documentos coletados pode ser realizada por uma nova implementação do módulo analisador (Seção 3.3.1).

A totalidade das características está imbuída no coletor ou pode ser dinamicamente adicionada a esse. É importante perceber que, desse total, 43.75% das características são nativamente suportadas pelo coletor enquanto 56.35% podem ser dinamicamente incorporadas.

Os dados mostram que o coletor, quando se leva em conta as características desejáveis a um coletor flexível, é maleável. Ademais, percebe-se que esse é coeso, com suas fronteiras bem definidas já que incorpora um conjunto pequeno de características e delega a maior parte delas para construções de terceiros. É importante notar que os percentuais podem variar, com implementações de terceiros agregadas ao núcleo do coletor e vice-versa.

5 CONCLUSÕES

O presente trabalho detalhou os conceitos necessários à construção de um coletor de dados flexível, que realize bom uso de recursos físicos como memória principal, ciclos de CPU e banda de Internet, e que possa ser operacionalizado como parte de outros sistemas.

Junto com uma sucinta introdução sobre o tema coleta de dados, foram definidos o problema de pesquisa, os objetivos e a metodologia empregada no curso do trabalho, além de delineada a fundamentação teórica que serviu de suporte ao desenvolvimento da solução final.

A fundamentação teórica realizou uma busca exaustiva na literatura acadêmica por trabalhos sobre coletores de dados, com artigos que detalham a concepção de uma arquitetura genérica para a coleta de dados, a coleta de dados focada, a coleta eficiente, a coleta distribuída, a polidez no trato de fontes servidoras de dados, a sincronização dos dados coletados para que espelhem com acurácia o estado das fontes de informação utilizadas na coleta, os aspectos cruciais à concepção de coletores com desempenho comparável aos descritos pela literatura, os mecanismos relacionados à tolerância a falhas, e o entendimento de múltiplos protocolos e tipos de arquivos pelo coletor.

O projeto do coletor sucedeu a fundamentação teórica, com a composição dos diagramas que delinearam a materialização da arquitetura flexível para a coleta de dados em um sistema computacional. Foram diagramados os requisitos e casos de uso de sistema, além de projetados os diagramas de classe e de sequência dos componentes.

Os produtos gerados pelo projeto foram insumo para a construção de um coletor de dados flexível que materializou os conceitos expostos na fundamentação teórica. O coletor foi então executado através de cenários de coleta cuidadosamente selecionados para a exposição de sua maleabilidade frente a distintos problemas de coleta. Os dados acerca da execução foram salvos e utilizados para mensurar e comparar o desempenho do coletor com outros coletores descritos pela literatura acadêmica. Ainda, uma listagem de características desejáveis a coletores flexíveis foi construída e cada item verificado no coletor construído.

A avaliação do coletor através da execução dos cenários e da verificação das características relevantes a coletores flexíveis mostrou que os **objetivos** do trabalho foram alcançados.

Os objetivos iniciaram com a **pesquisa e análise das diferentes soluções para a coleta de dados**. Das dezenas de descrições de coletores encontrados na literatura acadêmica, oito foram selecionadas e referenciadas na área para trabalhos correlatos, a Seção 2.4.

Também foram **compreendidas e analisadas as características necessárias a realização de um coletor extensível**. A fundamentação teórica, Capítulo 2, está permeada por trabalhos que apresentam tais características. A seção do projeto (Capítulo 3), com seus requisitos, diagramas de casos de uso, classe e sequência, formalizam a compreensão das características para que pudessem ser materializadas em um coletor. Ainda, a Seção 4.1.2 mostra uma lista de características desejáveis a coletores flexíveis que foi de suma importância a concepção da arquitetura flexível e também do coletor.

A **modelagem conceitual da arquitetura** foi também realizada. Ela é evidenciada no projeto (Capítulo 3), através dos vários diagramas dispostos. Vale ressaltar que a arquitetura derivou da proposta apresentada na Figura 3.

A modelagem conceitual foi **materializada através de um sistema computacional** utilizando a linguagem Java. Como o coletor está em conformidade com a arquitetura flexível para coleta proposta pelo trabalho, foi esse o objeto de análise utilizado para indicar se a arquitetura é ou não maleável.

O **teste e avaliação da implementação da arquitetura** foram realizados na Seção 4.1 através de três etapas: execução de cenários ecléticos de coleta para identificar a capacidade do coletor de recuperar dados em diferentes contextos, a checagem de determinadas características desejáveis a coletores flexíveis, e também a análise do desempenho do coletor.

A **documentação dos desenvolvimentos e resultados** foi exaustivamente realizada pelo presente trabalho. Os resultados mostraram que a característica flexibilidade está presente no coletor já que suas propriedades foram verificadas com sucesso contra uma lista de características desejáveis a coletores flexíveis. Também os cenários ecléticos de coleta foram executados com facilidade, mais uma demonstração da flexibilidade da arquitetura proposta. Ainda, o coletor se mostrou leve, contendo não mais do que algumas classes e interfaces escritas em Java e suportadas por apenas três bibliotecas de terceiros.

5.1 Trabalhos Futuros

Espaços para novos desdobramentos da pesquisa apresentada por este trabalho foram identificados no curso do TCC. Alguns desses espaços não foram endereçados pela restrição de recursos como o tempo enquanto outros não compunham o foco da pesquisa e seriam mais bem acomodados através de novos estudos. Tais espaços são apresentados como trabalhos futuros até o final desta seção.

A avaliação da flexibilidade da arquitetura de coleta, a principal característica da proposta, não foi exaustivamente executada. Enquanto os três cenários apresentados (Seção 4.1.1) estressaram alguns aspectos relacionados a coletores flexíveis, outros aspectos citados no curso do texto não foram demonstrados na prática. Um exemplo é a autenticação. Existe então uma abertura para que trabalhos de avaliação da flexibilidade da arquitetura proposta possam ser desenvolvidos.

A avaliação do desempenho da arquitetura, através do emprego do software coletor que a materializa na recuperação de documentos a partir da Web, se deu em um ambiente limitado. Um único computador foi utilizado, pouca banda para o acesso à Internet foi disponibilizada e a janela de tempo usada na execução do coletor foi demasiadamente curta. Para um real entendimento do desempenho do coletor, e para que a comparação com outros coletores seja adequada, é preciso executar a arquitetura com recursos de hardware mais generosos.

Apesar de a arquitetura apresentada prever a distribuição do processo de coleta entre computadores distintos pela criação de um novo executor (Seção 3.3.7), a materialização desse item foi ignorada. O motivo foi o alto custo para a implementação da solução distribuída em face do pequeno retorno para o a demonstração do requisito flexibilidade. É possível então uma nova pesquisa que visione essa questão, a da distribuição do processo de coleta utilizando a arquitetura proposta.

Existem outras fontes mantenedoras de dados que não foram destacadas pelo presente trabalho. Por exemplo, a coleta de dados a partir de estruturas relacionais como bancos de dados compostos de registros alocados em tabelas interligadas não foi considerada na arquitetura e pode ser trabalhada em outros estudos.

Uma das principais características da arquitetura é a habilidade de absorver diferentes necessidades de coleta e retornar dados relevantes ao cliente do sistema coletor. Para a captura das necessidades de coleta na presente arquitetura, o cliente necessita configurar um conjunto de

módulos, muitas vezes escrever novos em conformidade com interfaces pré-definidas, para que o coletor saiba quais documentos recuperar. Ontologias poderiam ser empregadas para descrever o problema de coleta do cliente. As ontologias seriam repassadas ao coletor que raciocinaria sobre elas na busca por documentos relevantes. Nesse caso, as ontologias são vistas como uma potencial melhor maneira de absorver diferentes necessidades de coleta, adicionando elementos semânticos ao sistema.

REFERÊNCIAS BIBLIOGRÁFICAS

AGGARWAL, Charu C.; AL-GARAWI, Fatima; YU, Philip S.. Intelligent crawling on the World Wide Web with arbitrary predicates. **Proceedings of the 10th International Conference on World Wide Web**, Hong Kong, p.96-105, 2001.

SILVA, Altigran S. da et al. CoBWeb A Crawler for the Brazilian Web. **Proceedings of the String Processing and Information Retrieval Symposium & International Workshop on Groupware, Washington**, p.184-184, 1999.

ANGKAWATTANAWIT, Niran; RUNGSAWANG, Arnon. Learnable Crawling: An Efficient Approach to Topic-specific Web Resource Discovery. **The 2nd International Symposium on Communications and Information Technology**, 2002.

ARASU, A. et al. Searching the Web. **ACM Transactions on Internet Technology,** New York, p. 2-43. 2001.

ARMITAGE, P.; BERRY, G.; MATTHEWS, J.N.S. **Statistical Methods in Medical Research.** United Kingdom: Blackwell Science Ltd, 2002.

BARBOSA, L.; FREIRE, J. An adaptive crawler for locating hidden-Web entry points. **Proceedings of the 16th International Conference on World Wide Web,** Banff, p. 441-450. 2007.

BEITZEL, Steven M. et al. Evaluation of filtering current news search results. **Proceedings of the 27th Annual International ACM Sigir Conference on Research and Development in Information Retrieval**, Sheffield, p.494-495, 2004.

BERNERS-LEE, T. **Uniform Resource Locators.** Disponível em <http://www.ietf.org/rfc/rfc1738.txt>. Acesso em: 23 junho 2009.

BRANDMAN, O. et al. Crawler-Friendly Web Servers. **Acm Sigmetrics Performance Evaluation Review,** New York, p. 9-14. 2000.

BRIN, S.; PAGE, L. The anatomy of a large-scale hypertextual Web search engine. **Computer Networks and ISDN Systems,** Brisbane, p. 107-117. abr. 1998.

BOLDI, P. et al. UbiCrawler: a scalable fully distributed Web crawler. **Software: Practice and Experience,** p. 711-726. mar. 2004.

BOSWELL, Dustin. Distributed high-performance web crawlers: a survey of the state of the art. **Department Of Electrical & Computer Engineering, University Of California**, San Diego, 2003.

BROKEL, J.M.; HARRISON, M.I. Redesigning care processes using an electronic health record: a system's experience. **Joint Commission Journal on Quality and Patient Safety,** United States, p. 82-92. fev. 2009.

CHAKRABARTI, BERG, S.M.V.D.; DOM, B. Focused crawling: a new approach to topic-specific Web resource discovery. **Computer Networks,** Toronto, p. 1623-1640. maio 1999.

CHAKRABARTI, Soumen; PUNERA, Kunal; SUBRAMANYAM, Mallela. Accelerated focused crawling through online relevance feedback. **Proceedings of the 11th International Conference on World Wide Web**, Honolulu, p.148-159, 2002.

CHAU, D.H. et al. Parallel crawling for online social networks. **Proceedings of the 16th International Conference on World Wide Web,** Banff, p. 1283-1284. 2007.

CHO, J.; GARCIA-MOLINA, H. Effective page refresh policies for Web crawlers. **Acm Transactions on Database Systems,** New York, p. 390-426. 2003a.

CHO, J.; GARCIA-MOLINA, H. Estimating frequency of change. **Acm Transactions on Internet Technology,** New York, p. 256-290. 2003b.

CHO, J.; GARCIA-MOLINA, H. Parallel crawlers. **Proceedings of the 11th International Conference on World Wide Web,** Honolulu, p. 124-135. 2002.

CHO, J.; GARCIA-MOLINA, H. Synchronizing a database to improve freshness. **Acm Sigmod Record,** Dallas, p. 117-128. 2000.

CHO, J.; GARCIA-MOLINA, H.; PAGE, L. Efficient crawling through URL ordering. **Proceedings of the Seventh International World Wide Web Conference,** Brisbane, p. 124-135. jun. 1998.

CHO, J.; NTOULAS, A. Effective change detection using sampling. **Proceedings of the 28th International Conference on Very Large Data Bases,** Hong Kong, p. 514-525. 2002.

CHO, J.; ROY, S. Impact of search engines on page popularity. **Proceedings of the 13th International Conference on World Wide Web**, New York, p.20-29, 2004.

CHO, Junghoo; SHIVAKUMAR, Narayanan; GARCIA-MOLINA, Hector. Finding replicated Web collections. **Acm Sigmod Record**, p.355-366, 2000.

DONG, H.; HUSSAIN, F.K.; CHANG, E.. A Transport Service Ontology-based Focused Crawler. **Fourth International Conference on Semantics, Knowledge and Grid, 2008. Skg '08.**, Beijing, p.49-56, 2007.

DONG, H.; HUSSAIN, F. K.; CHANG, E.. State of the Art in Semantic Focused Crawlers. **Lecture Notes In Computer Science: Proceedings of the International Conference on Computational Science and Its Applications: Part II,** Seoul, p. 910-924. 2009.

EDWARDS, J.; MCCURLEY, K.; TOMLIN, J.. An adaptive model for optimizing performance of an incremental web crawler. **Proceedings of the 10th International Conference on World Wide Web,** Hong Kong, p. 106-113. 2001.

EHRIG, M.; MAEDCHE, A. Ontology-focused crawling of Web documents. **Proceedings of the 2003 Acm Symposium On Applied Computing,** Melbourne, p. 1174-1178. 2003.

ERIKSSON, Hans-erik et al. **UML 2 Toolkit.** Indianapolis: Wiley Publishing, 2004. 552 p.

FIELDING, R. et al. **Hypertext Transfer Protocol – HTTP/1.1.** Disponível em <http://www.w3.org/Protocols/rfc2616/rfc2616.html >. Acesso em: 03 novembro 2009.

FOSTER, Ian et al. A security architecture for computational grids. **Proceedings of the 5th Acm Conference on Computer and Communications Security**, San Francisco, p.83-92, 1998.

GAMMA, Erich et al. **Design Patterns: Elements of Reusable Object-Oriented Software.** Addison-wesley Professional, 1994. 416 p.

GAO, Weizheng; LEE, Hyun Chul; MIAO, Yingbo. Geographically focused collaborative crawling. **Proceedings of the 15th International Conference on World Wide Web**, Edinburgh, p.287-296, 2006.

GIRARDI, Christian; RICCA, Filippo; TONELLA, Paolo. Web Crawlers Compared. **International Journal of Web Information Systems**, v. 2, n. 2, p.85-94, 2006.

GOMES, D.; SILVA, M.J. The Viúva Negra crawler: an experience report. **Software: practice and Experience,** New York, p. 161-188. fev. 2008.

GULLI, A.. He anatomy of a news search engine. **Special Interest Tracks and Posters of the 14th International Conference on World Wide Web**, Chiba, p.880-881, 2005.

HABIB, Ahsan; ABRAMS, Marc. Analysis of Sources of Latency in Downloading Web Pages. **Proceedings of Webnet World Conference on the WWW and Internet 2000**, Chesapeake, p.227-232, 2000.

HAFRI, Y.; DJERABA, C.. High performance crawling system. **Proceedings of the 6th Acm Sigmm International Workshop on Multimedia Information Retrieval,** New York, p. 299-306. 2004.

HEYDON, A.; NAJORK, M. Mercator: A scalable, extensible Web crawler. **World Wide Web,** Toronto, p. 219-229. dez. 1999.

HENZINGER, Monika et al. Query-Free News Search. **World Wide Web**, v. 8, n. 2, p.101-126, 2005.

KORFHAGE, Robert R.. Information storage and retrieval. Estados Unidos da América: **Wiley Computer Pub.,** 1997. 368 p.

KOSTER, M. **A standard for robot exclusion.** Disponível em <www.robotstxt.org/wc/norobots.html>. Acesso em: 23 junho 2009.

KOZANIDIS, Lefteris. An Ontology-Based Focused Crawler. **Natural Language and Information Systems**, London, v. 2008, n. 5039, p.376-379, 2008.

KRITIKOPOULOS, Apostolos; SIDERI, Martha; STROGGILOS, Kostantinos. CrawlWave: A Distributed Crawler. **3rd Hellenic Conference on Artificial Intelligence**, Pythagorion, 2004.

LAWRENCE, S.; GILES, C.L. Searching the World Wide Web. **Science**, p. 98-100. abr. 1998.

LEE, D. et al. LeeDeo: Web-Crawled Academic Video Search Engine. **Ism: Proceedings of the 2008 Tenth IEEE International Symposium on Multimedia,** Washington, p. 497-502. 2008.

LI, J.; FURUSE, K.; YAMAGUCHI, K. Focused crawling by exploiting anchor text using decision tree. **Special Interest Tracks and Posters of the 14th International Conference on World Wide Web,** Chiba, p. 1190-1191. 2005.

LIU, B. **Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data.** 2. ed. Springer, 2009. 532 p.

LUONG, H. P.; GAUCH, S.; WANG, Qiang. Ontology-Based Focused Crawling. **International Conference on Information, Process, and Knowledge Management, 2009. Eknow '09.**, Cancun, p.123-128, 2009.

MAYER, Ulrich. Protein Information Crawler (PIC): Extensive spidering of multiple protein information resources for large protein sets. **Proteomics**, p.42-44, 2007.

MCCOWN, Frank; NELSON, Michael L.. Evaluation of crawling policies for a web-repository crawler. **Proceedings of the Seventeenth Conference on Hypertext and Hypermedia**, Odense, p.157-168, 2006.

MOSHCHUK, Alexander et al. FindingA Crawler-based Study of Spyware on the Web. **Proceedings of the 2006 Network and Distributed System**, San Diego, p.17-33, 2006.

NTOULAS, A.; CHO, J.; OLSTON, C. What's new on the web?: the evolution of the web from a search engine perspective. **Proceedings of the 13th International Conference on World Wide Web,** New York, p. 1-12. 2004.

NTOULAS, Alexandros; ZERFOS, Petros; CHO, Junghoo. Downloading textual hidden web content through keyword queries. **Proceedings of the 5th Acm/ieee-cs Joint Conference on Digital Libraries**, Denver, p.100-109, 2005.

PAHAL, N.; CHAUHAN, N.; SHARMA, A.k.. Context-Ontology Driven Focused Crawling of Web Documents. **Wireless Communication and Sensor Networks, 2007. Wcsn**, Allahabad, p.121-124, 2007.

PANT, G.; BRADSHAW, S.; MENCZER, F. Search Engine-Crawler Symbiosis: Adapting to Community Interests. **Research and Advanced Technology For Digital Libraries,** p. 221-232. 2003.

PANT, Gautam; SRINIVASAN, Padmini; MENCZER, Filippo. Exploration versus Exploitation in Topic Driven Crawlers. **WWW02 Workshop on Web Dynamics**, Honolulu, 2002.

RAGHAVAN, S.; GARCIA-MOLINA, H. Crawling the Hidden Web. **Proceedings of the 27th International Conference on Very Large Data Bases,** San Francisco, p. 129-138. 2001.

SIA, Ka Cheung; CHO, Junghoo; CHO, Hyun-kyu. Efficient Monitoring Algorithm for Fast News Alerts. **Ieee Transactions on Knowledge and Data Engineering,**, Los Angeles, p.950-961, 2007.

SILVA, A.S. et al. CoBWeb - A crawler for the Brazilian Web. **String Processing and Information Retrieval Symposium,** Cancun, p. 184-191. ago. 2002.

SHKAPENYUK, V.; SUEL, T. Design and implementation of a high-performance distributed Webcrawler. **Proceedings. 18th International Conference on Data Engineering,** San Jose, p. 357-368. 2002.

SRINIVASAN, P.; MENCZER, F.; PANT, G. A General Evaluation Framework for Topical Crawlers. **Information Retrieval,** p. 417-447. 04 abr. 2005.

STAMATAKIS, Konstantinos et al. Domain-Specific Web Site Identification: The CROSSMARC Focused Web Crawler. **Proceedings of the Second International Workshop on Web Document Analysis**, 2003.

SU, Chang et al. An Efficient Adaptive Focused Crawler Based on Ontology Learning. **Proceedings of the Fifth International Conference on Hybrid Intelligent Systems**, p.73-78, 2005.

TANG, T.T. et al. Focused crawling for both topical relevance and quality of medical information. **Proceedings of the 14th Acm International Conference on Information and Knowledge Management,** Bremen, p. 147-154. 2005.

WAN, Yuan; TONG, Hengqing. URL Assignment Algorithm of Crawler in Distributed System Based on Hash. **International Conference on Networking, Sensing and Control, 2008. Icnsc 2008. Ieee**, Sanya, p.1632-1635, 2008.

WANG, C. et al. On-line topical importance estimation: an effective focused crawling algorithm combining link and content analysis. **Journal of Zhejiang University: Science A,** Hangzhou, p. 1114-1124. ago. 2009.

XU, Q.; ZUO, W. First-order focused crawling. **Proceedings of the 16th International Conference on World Wide Web,** Banff, p. 1159-1160. 2007.

YADAV, D.; SHARMA, A. K.; GUPTA, J. P.. Topical web crawling using weighted anchor text and web page change detection techniques. **Wseas Transactions on Information Science and Applications,** Stevens Point, p. 263-275. fev. 2009.

ZHENG, Hai-tao; KANG, Bo-yeong; KIM, Hong-gee. Learnable Focused Crawling Based on Ontology. **Information Retrieval Technology**, Harbin, v. 2008, n. 4993, p.264-275, 2008.

ZHU, Kunpeng et al. A Full Distributed Web Crawler Based on Structured Network. **Information Retrieval Technology**, Harbin, p.478-483, 2008.

APÊNDICES

A INTERFACES DOS MÓDULOS DO COLETOR EM JAVA

ANALISADOR

```
package net.sourceforge.retriever.analyzer;

import java.util.Map;

import net.sourceforge.retriever.fetcher.Resource;

/**
 * <p>
 * Analyzers are used to consume crawled information.
 * </p>
 *
 * <p>
 * In order to do that, users must give this class an implementation
 * and pass it to the <code>Crawler</code> object using the
 * <code>addAnalyzer(Analyzer)</code> method.
 * </p>
 *
 * <p>
 * This way, users will be able to act upon every crawled resource in
 * order to achieve their goals. For instance, to persist data into
 * a database or inverted index, to print some information on the console,
 * to log data, and so forth.
 * </p>
 *
 * <p>A special care must be taken while developing your own
 * <code>Analyzer</code> since each instance is likely to be accessed by
 * multiple threads at the
 * same time.</p>
 *
 * @see net.sourceforge.retriever.Retriever#addAnalyzer(Analyzer)
 */
public interface Analyzer {


        /**
         * Operation that let users consume crawled resources.
         *
         * @param resource The resource to be consumed.
         * @param additionalInfo TODO
         */
        void analyze(Resource resource, Map<String, Object> additionalInfo);
}
```

AUTENTICADOR

```
package net.sourceforge.retriever.authentication;

import java.io.InputStream;
import java.net.URL;

import net.sourceforge.retriever.Retriever;
import net.sourceforge.retriever.feedback.Feedback;

/**
 * <p>
 * This interface defines the behavior of the authentication module. It
 * allows resources reachable only after some login process to be crawled.
 * For instance, various web pages have a login form, requiring a user name
 * and password for clients to access specific pages. This module allows
 * those specific pages to be crawled.
 * </p>
 *
 * <p>
 * There are different kinds of authentication mechanisms, and each class
 * implementing this interface is likely to be addressing one of them.
 * </p>
 *
 * @see Retriever#setAuthentication(Authentication)
 */
public interface Authentication {


        /**
         * <p>
         * Operation used to authenticate the crawler, so it will be able to
         * collect resources that wouldn't be crawled otherwise. It takes an
         * URL and its content, which may be used in the authentication
         * process.
         * </p>
         *
         * @param url The URL where authentication will occur.
         * @param urlContent The content of the URL, that can be used to aid
         *        the authentication process.
         * @return An URL to the page returned by the authentication.
         */
        URL authenticate(URL url, InputStream inputStream);

        /**
         * The Feedback module.
         *
         * @param feedback
         */
        void setFeedback(Feedback feedback);
}
```

FEEDBACK

```
package net.sourceforge.retriever.feedback;

/**
 * <p>
 * Interface that allows the crawler to provide feedback about what is going
 * on during the crawling process.
 * </p>
 *
 * <p>
 * For users to consume these feedbacks, an appropriate implementation of
 * this interface must be written and passed to the crawler.
 * </p>
 *
 * @see net.sourceforge.retriever.Retriever#Crawler(CrawlerFeedback)
 */
public interface Feedback {


        /**
         * Used for error messages.
         *
         * @param url States that the error message is associated with a URL.
         * @param message The message.
         * @param caught The exception.
         */
        void error(String url, String message, Throwable caught);

        /**
         * Used for warning messages.
         *
         * @param url States that the warning message is associated with a URL.
         * @param message The message.
         */
        void warning(String url, String message);

        /**
         * Used for informative messages.
         *
         * @param url States that the info message is associated with a URL.
         * @param message The message.
         */
        void info(String url, String message);
}
```

EXECUTOR

```
package net.sourceforge.retriever.executor;

import net.sourceforge.retriever.analyzer.Analyzer;
import net.sourceforge.retriever.authentication.Authentication;
import net.sourceforge.retriever.feedback.Feedback;
import net.sourceforge.retriever.fetcher.dns.DNSResolver;
import net.sourceforge.retriever.filter.Filter;
import net.sourceforge.retriever.frontier.Frontier;

/**
 * Executes a single step in the crawling process.
 *
 * This component is called multiple times until a crawling stop criteria
 * is reached.
 */
public interface Executor {


        /**
         * Creates an executor.
         */
        Executor create();

        /**
         * Checks if the executor is running.
         */
        boolean isExecuting();

        /**
         * Shutdows the executor.
         */
        void shutdown();

        /**
         * Wires together all other crawler components to execute a single
         * crawling step.
         */
        void execute(Frontier urlFrontier, Filter filter, Analyzer analyzer,
                        Feedback feedback, Authentication authentication,
                        Object emptyFrontierLock, DNSResolver dnsResolver);
}
```

FILTROO

```java
package net.sourceforge.retriever.filter;

import net.sourceforge.retriever.fetcher.Resource;

/**
 * <p>
 * Filters are used to filter resources during the crawling process.
 * </p>
 *
 * <p>
 * It has two moments. The first one happens before resources are fetched,
 * when they are nothing more than their URLs. In this stage, one can inspect
 * the URL and decided if it's worth crawling or not.
 * </p>
 *
 * <p>
 * The other moment is used to filter resources after they are fetched. In
 * this stage, the resource contains not only the URL, but also some other
 * useful information like the content-type, the content, length, etc. Users
 * can, during this step, filter resources by a bigger set of information.
 * </p>
 */
public interface Filter {


        /**
         * Filter a resource before the fetching process, when it's only
         * a URL.
         *
         * @param url The URL used in the filter process.
         * @return True if the URL was accepted and won't be filtered and
         *         false otherwise.
         */
        boolean acceptBeforeFetch(final String url);

        /**
         * Filter a resource after the fetching process, when it's much
         * richer than a simple URL, holding information like content-type,
         * the content itself, and so on.
         *
         * @param url The resource used in the filter process.
         * @return True if the resource was accepted and won't be filtered
         *         and false otherwise.
         */
        boolean acceptAfterFetch(final Resource resource);
}
```

FRONTIER

```
package net.sourceforge.retriever.frontier;

import java.util.List;

/**
 * Holds the list of URLs to be crawled.
 */
public interface Frontier {


        /**
         * Enqueues a URL wrapped by a FrontierUrl object.
         */
        void enqueue(FrontierUrl frontierURL);

        /**
         * Dequeues a list of URLs to be crawled.
         */
        List<FrontierUrl> dequeue();

        /**
         * Used by frontier clients to notify that URL from some IP
         * were already crawled.
         */
        void notifyFetchedIp(String ip);

        /**
         * If the frontier has URLs.
         */
        boolean hasURLs();

        /**
         * Removes all URLs from the frontier.
         */
        void reset();

        /**
         * Intervall between fetches for all URLs sharing the same IP.
         */
        void setIntervalBetweenFetchesInMillis(final long time);

        /**
         * Interval between fetches for all URLs from an specific IP.
         */
        void setIntervalBetweenFetchesInMillis(final String ip,
                        final long time);
}
```

FETCHER

```
package net.sourceforge.retriever.fetcher;

import java.io.InputStream;

/**
 * <p>
 * A fetcher receives an URL and produces an object holding information about
 * the resource.
 * </p>
 */
public abstract class Fetcher {

    public abstract boolean canCrawl();

    protected abstract List<String> getInnerURLs(String charset);

    protected abstract void treatFetchException(
            final URLConnection urlConnection, final Exception e);

    protected abstract void close(URLConnection urlConnection);

    protected abstract void lastFetch(URLConnection urlConnection);

    public Fetcher(final URL url) {
        // Suppressed due to space constraints.
    }

    public static Fetcher create(final URL url) throws Exception {
        // Suppressed due to space constraints.
        return null;
    }

    public Resource fetch() throws Exception {
        // Suppressed due to space constraints.
        return null;
    }

    public Resource fetch(final boolean lastFetch) throws Exception {
        // Suppressed due to space constraints.
        return null;
    }

    public void resetInputStream() {
        // Suppressed due to space constraints.
    }

    public static void setFetchTimeout(final int milliseconds) {
        // Suppressed due to space constraints.
    }

    public static int getFetchTimeout() {
        // Suppressed due to space constraints.
        return 0;
    }

    public static void setConnectTimeout(final int milliseconds) {
        // Suppressed due to space constraints.
    }

    public static int getConnectTimeout() {
        // Suppressed due to space constraints.
        return 0;
    }

    public static void register(final String protocol,
            final Class<? extends Fetcher> resource) {
        // Suppressed due to space constraints.
    }

    protected URL getURL() {
        // Suppressed due to space constraints.
        return null;
    }

    protected InputStream getInputStream() {
        // Suppressed due to space constraints.
        return null;
    }
}
```

B CÓDIGO JAVA PARA A COLETA LIVRE NA WEB

```
package br.univali.tcc.webcrawler;

import java.net.MalformedURLException;

public class WebCrawler {

    public static void main(final String[] args) throws MalformedURLException,
            SQLException, ClassNotFoundException {
        new WebCrawler().start();
    }

    public void start() throws MalformedURLException, SQLException,
            ClassNotFoundException {
        final Retriever retriever = new Retriever(new FeedbackStorer());
        retriever.setExecutor(new ThreadExecutor(100, 100, 60,
                TimeUnit.SECONDS, 1000));
        retriever.setFrontier(new LimitFrontierSize(
                new OneTimeOnlyDataBaseFrontier(new PoliteFrontier())));
        retriever.addSeed(new URL("http://www.uol.com.br/"));
        retriever.addSeed(new URL("http://www.globo.com/"));
        retriever.addSeed(new URL("http://www.yahoo.com.br/"));
        retriever.setAnalyzer(new CrawledDataStorer());
        retriever.start();
    }

    private class CrawledDataStorer implements Analyzer {

        private volatile int numberOfDocuments;

        @Override
        public void analyze(final Resource resource,
                final Map<String, Object> additionalInfo) {
            Persistence.insertDocument(resource.getURL().toExternalForm(),
                    resource.getContentType(), resource.getLength(), null);
            System.out.println(++this.numberOfDocuments);
        }
    }

    private class FeedbackStorer implements Feedback {

        @Override
        public void error(final String url, final String message,
                final Throwable caught) {
            Persistence.insertDocument(url, null, 0, caught.getMessage());
        }

        @Override
        public void info(final String url, final String message) {
        }

        @Override
        public void warning(final String url, final String message) {
        }
    }
}
```

C CÓDIGO JAVA PARA A DESCOBERTA DE FEEDS

```
package br.univali.tcc.feedcrawler;

import java.io.IOException;

public class FeedDiscover {

    public static void main(final String[] args) throws MalformedURLException {
        new FeedDiscover().start();
    }

    public void start() throws MalformedURLException {
        final Retriever retriever = new Retriever();
        this.addSeeds(retriever);
        this.setFilter(retriever);
        retriever.setAnalyzer(new FeedStorer());
        retriever.start();
    }

    private void addSeeds(final Retriever retriever)
            throws MalformedURLException {
        retriever.addSeed(new URL("http://www.time.com/"));
        retriever.addSeed(new URL("http://www.nytimes.com/"));
        retriever.addSeed(new URL("http://www.folha.uol.com.br/"));
    }

    private void setFilter(final Retriever retriever) {
        final URLRegexFilter filter = new URLRegexFilter();
        filter.addRegex(".*time\\.com.*");
        filter.addRegex(".*nytimes\\.com.*");
        filter.addRegex(".*folha\\.uol\\.com\\.br.*");
        retriever.setFilter(filter);
    }

    private class FeedStorer implements Analyzer {

        @Override
        public synchronized void analyze(final Resource resource,
                final Map<String, Object> additionalInfo) {
            final String contentType = resource.getContentType();
            if ((contentType != null && (contentType.contains("atom")
                    || contentType.contains("rss") || contentType
                    .contains("xml")))
                    || resource.getURL().toExternalForm().endsWith("xml")) {

                InputStreamReader reader = null;
                try {
                        reader = new InputStreamReader(resource.getURL()
                                .openStream());
                        new SyndFeedInput().build(reader);
                        Persistence.insertFeed(resource.getURL()
                                .toExternalForm());
                } catch (Exception e) {
                } finally {
                    if (reader != null) {
                        try {
                            reader.close();
                        } catch (final IOException e) {
                            e.printStackTrace();
                        }
                    }
                }
            }
        }
    }
}
```

D CÓDIGO JAVA PARA A COLETA EM LANS

```
package br.univali.tcc.crawler.lan;

import java.net.MalformedURLException;

public class LanCrawler {

    public void start() throws MalformedURLException, InterruptedException {
        final Retriever retriever = new Retriever();
        this.addSeeds(retriever);
        retriever.setAnalyzer(new LanAnalyzer());
        retriever.start();
    }

    private void addSeeds(Retriever retriever) throws InterruptedException,
            MalformedURLException {
        final Set<String> ips = new ComputersDiscovery().discover();
        for (String ip : ips) {
            retriever.addSeed(new URL("smb://" + ip + "/"));
        }
    }

    public static void main(final String[] args) throws MalformedURLException,
            InterruptedException {
        new LanCrawler().start();
    }

    private class LanAnalyzer implements Analyzer {

        private volatile int i;

        @Override
        public void analyze(final Resource resource,
                final Map<String, Object> additionalInformation) {
            System.out
                    .println(++i + " - " + resource.getURL().toExternalForm());
            Persistence.insertLanDocument(resource.getURL().toExternalForm(),
                    resource.getContentType(), resource.getLength(), null);
        }
    }
}
```