LARGE-SCALE ELASTIC ARCHITECTURE FOR DATA AS A SERVICE

Project Number:	FP7-ICT-318809
Project Title:	LEADS: Large-Scale Elastic Architecture for Data as a Service
Deliverable Number:	D1.2
Title of Deliverable:	Real-time content discovery through on-the-fly publishing
Contractual Date of Delivery:	M12 – 9/30/2013
Actual Date of Delivery:	9/30/2013

Abstract

This deliverable presents the user-aided data collection mechanism of LEADS and focuses on the design and implementation details of the user-aided publishing interface. It also presents the experimental evaluation of the ability of user-aided publishing for collecting data compared to the crawling-based data collection. Finally, it presents an initial prototype implementation of the distributed crawlers.

List of Contributors

Name	Organization	E-mail
Xiao Bai	BM-Y!	xbai@yahoo-inc.com
Matthieu Morel	BM-Y!	matthieu@yahoo-inc.com
Ata Turk	BM-Y!	ata@yahoo-inc.com
Pascal Felber	UniNE	Pascal.Felber@unine.ch
Marcelo Pasin	UniNE	Marcelo.Pasin@unine.ch
Etienne Rivière	UniNE	Etienne.Riviere@unine.ch
Pierre Sutra	UniNE	Pierre.Sutra@unine.ch
Christof Fetzer	TUD	Christof.Fetzer@tu-dresden.de
André Martin	TUD	Andre.Martin@tu-dresden.de
Do Le Quoc	TUD	Do@se.inf.tu-dresden.de
Frezewd Lemma Tena	TUD	Frezewd_Lemma.Tena@mailbox.tu-dresden.de
Lenar Yazdanov	TUD	Lenar.Yazdanov@tu-dresden.de

Document Approval

	Name	Email	Date
Approved by WP Leader	Xiao Bai	xbai@yahoo-inc.com	2013-9-11
Approved by TSI	Minos Garofalakis	minos@acm.org	2013-9-20
Approved by AoTerra	Jens Struckmeier	jens.struckmeier@aoterra.de	2013-9-20

Contents

LIST OF CONTRIBUTORS II
DOCUMENT APPROVAL III
CONTENTS IV
LIST OF FIGURES V
EXECUTIVE SUMMARY 1
1. INTRODUCTION 2
2. DISTRIBUTED DATA COLLECTION IN LEADS 3
3. SPECIFICATION OF USER-AIDED DATA COLLECTION 4
3.1 HIGH-LEVEL DESIGN 4
3.2 REQUIREMENTS 5
3.2.1 Requirements on quality *6*
3.2.2 Requirements on efficiency *6*
4. DESIGN AND IMPLEMENTATION OF THE USER-AIDED PUBLISHING INTERFACE 7
4.1 FUNCTIONALITY OF THE USER-AIDED PUBLISHING INTERFACE 7
4.2 DESIGN OF THE USER-AIDED PUBLISHING INTERFACE 7
4.3 IMPLEMENTATION OF THE USER-AIDED PUBLISHING INTERFACE 11
4.3.1 Overview *11*
4.3.2 Implementation of key components *12*
4.4 INTERACTION WITH THE BACKEND 15
4.5 DEPLOYMENT OF THE USER-AIDED PUBLISHING INTERFACE 16
5. EVALUATION OF USER-AIDED DATA COLLECTION 17
5.1 EXPERIMENTAL SETUP 17
5.2 EXPERIMENTAL RESULTS 17
6. DESIGN AND IMPLEMENTATION OF THE WEB CRAWLER 19
6.1 SPECIFICATION OF CRAWLING-BASED DATA COLLECTION 19
6.1.1 Requirements on quality *20*
6.1.2 Requirements on efficiency *21*
6.1.3 Requirements on cost *21*
6.1.4 Additional requirements *22*
6.2 DESIGN OF A WEB CRAWLER 22
6.3 CURRENT IMPLEMENTATION OF THE PROTOTYPE CRAWLING-BASED DATA COLLECTION 23
6.3.1 Capabilities *24*
6.3.2 Components *24*
6.3.3 Internal workflow for Use Case 1 .... *24*
7. CONCLUSION 25
8. REFERENCES 25

List of Figures

Figure 1: Architecture of the distributed data collection framework 3
Figure 2: Design of user-aided data collection framework in a micro-cloud 5
Figure 3: User-aided publishing interface of LEADS 8
Figure 4: Popup publishing window 8
Figure 5: Example of different scrape modes 9
Figure 6: Example of “Content overview” field 10
Figure 7: Status of user-aided publishing 11
Figure 8: Manifest.json for the user-aided publishing interface 12
Figure 9: Options page of the user-aided publishing interface 13
Figure 10: Skeleton of options.js for the user-aided publishing interface 13
Figure 11: Skeleton of popup.js for the user-aided publishing interface 15
Figure 12: Skeleton of lead.js for the user-aided publishing interface 15
Figure 13: Format of JSON object published to LEADS 16
Figure 14: Implementation of the backend Key-Value store 16
Figure 15: Percentage of URLs that cannot be reached by crawling-based data collection 18
Figure 16: Percentage of URLs that can be published earlier by users 18
Figure 17: Architecture of the crawling-based data collection framework 20
Figure 18: Architecture of a web crawler 23

GLOSSARY

EU	European Union
FP7	Seventh Framework Programme
DaaS	Data-as-a-Service
URL	Universal Resource Locator
JSON	JavaScript Object Notation
HTTP	HyperText Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
CSS	Cascading Style Sheets

Executive summary

LEADS is a decentralized Data-as-a-Service (DaaS) framework that runs on an elastic collection of micro-clouds to gather, store and process data. The data collection in LEADS is performed in a fully distributed way through two mechanisms: crawling-based data collection, which is a geographically distributed version of the traditional web crawling, and user-aided data collection, which relies on LEADS users to actively report their data to the platform.

In this deliverable, we first present the architecture of the distributed data collection of LEADS. We then present the user-aided data collection mechanism of LEADS. The user-aided data collections aims to collect user generated data and data that are difficult to be collected by crawling. It innovates in the design of a user publishing interface that enables real-time publishing of user generated content and the application oriented data pre-processing that ensures only new and high quality data are incorporated to LEADS. We focus on the design and implementation of the user-aided publishing interface and leave the pre-processing of user published data to future deliverables.

Specifically, the user-aided data collection relies on users to provide data to LEADS. Users can send the web pages they create or they visit, along with their own tags or comments to the Key-Value store of LEADS. A key component for enabling this functionality is what we call the user-aided publishing interface, which allows users to either explicitly or implicitly send their data (as well as the associated metadata) to LEADS. The user-aided publishing interface is designed as a plug-in on web browsers, and is implemented as a Chrome extension. Therefore, users are able to use it easily on their end devices.

We also evaluate, using large-scale datasets collected both by crawling and by means comparable to user-aided data publishing, the potential of user-aided publishing to collect the data that is difficult to collect with crawlers. The result is very promising: a large fraction of pages collected from the users using user-aided publishing cannot be timely crawled by the crawlers.

Finally, we present the architecture of the crawling-based data collection mechanism of LEADS, and an initial simple implementation of its principles. The crawlers fetch the publicly available web pages in a similar way as the traditional web crawlers of search engines, but innovates in distributing the crawling process to geographically distributed micro-clouds while ensuring the efficiency and the low cost. We also present a prototype implementation of the distributed scrawlers. This crawler is built on top of flaxcrawler and interacts with Infinispan that implements the core of the LEADS storage layer.

1. Introduction

There is a wealth of publicly available data in today's Internet that can be exploited by large and small companies in various business domains. Collecting, storing, processing, and querying the rapidly increasing amounts of data is becoming a major challenge, and having the capability to do so is a strong asset for the few big companies with sufficient infrastructural resources. LEADS is designed as a decentralized Data-as-a-Service (DaaS) framework that runs on an elastic collection of micro-clouds to gather, store and process data so that small companies can perform large and challenging tasks in a pay-as-you-go approach using LEADS.

This work package focuses on the mechanisms of collecting data that is publically available in the Internet, which will form the basis of the service that LEADS aims to provide. There exist many types of public data that can be made available on LEADS for various application domains and companies. Web content, inter-connected Web graph, and Web content enriched by user-generated content are especially of interest for the LEADS platform. The gathered data can be served to users (small companies, research institutes, individual users etc.) to support various form of applications, such as (1) a Web graph service that allows users to access and analyse the inter-connected Web graph and the associated content, (2) data mining service that allows users to mine information (feedback, trends, etc.) from the available Web content stored in the platform, based on the real-time stream of collected content or a combination of both.

The data collection in LEADS is performed in a fully distributed way. LEADS uses a decentralized platform composed of a collection of micro-clouds, consisting of a number of servers in the double-digit range (typically between 12 and 24 multi-core servers). The data will be collected through the collaboration of these micro-clouds.

There exist many design alternatives for sequential [LLW+08], parallel [HN99, CG02, SS02, ZD02, BCS+04], and geographically distributed [CPJ+08, EMP+05, EMP+08] Web crawlers. The three main quality objectives are achieving high collection quality through download scheduling [CGP98, NW01], maintaining page freshness [CG00, CG03], and obtaining high Web coverage [DGK+07, LG00]. Unfortunately, with such forms of crawling, data collection is limited to following hyperlinks, and not all content on the Web can be accessed that way [Ber01]. For example, a high fraction of the content on the Web corresponds to the *deep web* that is only accessible through web forms [RGM01], which are often dynamically generated only when users fill the forms and submit them to the back-end databases. The main challenge to collect such data through crawling is to discover entry points to the hidden Web pages [BF07] or to sample them via automated form filling [JKK+08, NZC05]. However, these approaches are not efficient enough while they risk incurring too much burden on the sites hosting the pages. LEADS aims to collect such pages to expand its data collection built upon traditional crawling. Besides, the traditional web usually does not provide users the facility to tag or comment the data they visit on the Web, and then query the corresponding tags or comments they previously generated. Yet, the prevalence of the social networking systems like Facebook, Twitter, etc. and social tagging systems like Delicious, Flickr, etc. have demonstrated that users indeed have the need to express themselves and to exploit the data through personalization when accessing and exploiting Web data. User-generated data like tags, comments, etc. are thus another form of data that LEADS aims to collect in order to enrich the data mining service it provides, such as sentiment or trend analysis.

Therefore, LEADS innovates on a new form of data collection, which we call *user-aided data collection*, to gather the web pages that are difficult to be crawled by crawlers, and the user-generated data around the crawled web pages, as soon as they are visited or generated, i.e., user-aided. Specif-

ically, the user-aided data collection mechanism relies on a user-publishing interface, i.e., a plug-in to the web browser of each user, to collect such data. When user accesses or creates a web page, she can associate metadata (e.g., tags, comments, etc.) with the page that she is visiting through the LEADS publishing interface. Then the URL of the page, as well as its metadata, will be sent to the user-aided data collection system. After filtering redundant and spam content, the data will be stored to the Key-Value store of the LEADS platform.

In this deliverable, we focus on the design and implementation of the user-aided publishing interface that allows LEADS to collect web data as well as user-generated data in near real time (Section 3 and Section 4). The filtering of redundant and spam content will be addressed in the following deliverable (D1.3). In this deliverable, we also provide experimental evaluation to estimate the amount of the data that can be gathered by the user-aided data collection mechanism in addition to the crawling-based data collection mechanism (Section 5). Besides, LEADS also deploys an crawling-based data collection mechanism that crawls the web pages in a similar way as the traditional crawlers of search engines, but performs the crawling in geographically distributed micro-clouds. Although it is indicated in the description of work that the focus of this deliverable is on the user-aided data collection, we provide a specification of the crawling-based data collection and a description of the initial implementation we build during the first year of the project. We chose to work in advance on the subject in order to ease integration and testing. We provide an early and simple crawler that will be later evolved (or used as an inspiration) for the more complete crawlers due at the end of the project.

2. Distributed data collection in LEADS

Before going to the details of the user-aided user-publishing interface, we first give an overview of the entire distributed data collection framework of LEADS. In LEADS, public data that is available in the Internet is collected in a fully distributed way, through two different mechanisms: crawling-based data collection and user-aided data collection. Figure 1 shows the high level architecture of the data collection framework of LEADS, focusing on the data flows inside the system.

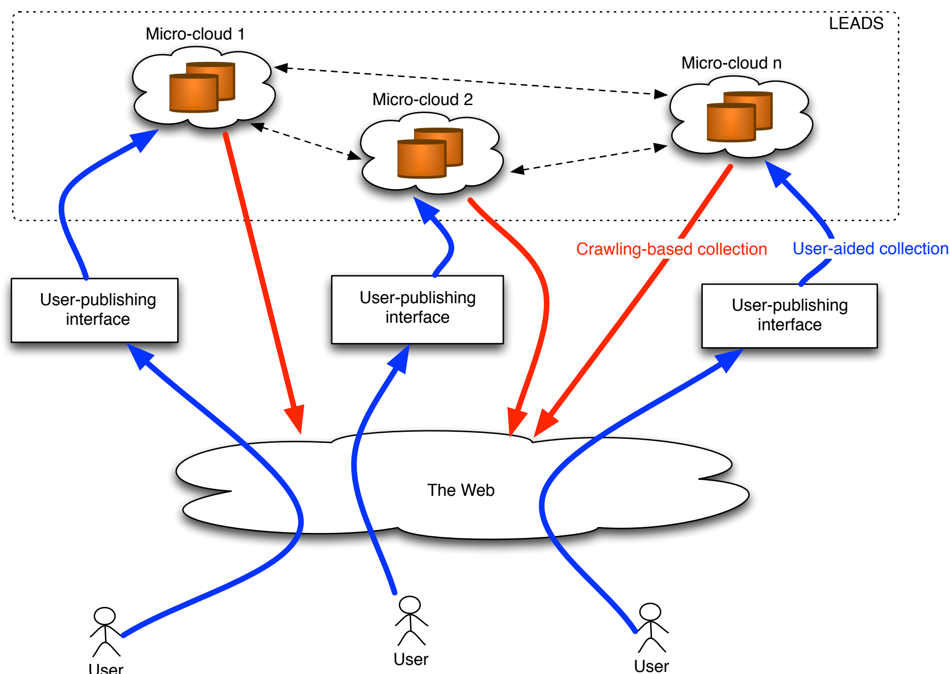


Figure 1: Architecture of the distributed data collection framework

The crawling-based data collection mechanism relies on web crawlers located in geographically distributed micro-clouds to fetch content from the Web. Each crawler is in charge of crawling a fraction of the web pages from the Web by following the link structure of the pages that have been fetched. Following the link structure is the standard technique used by commercial search engines to discover and acquire public data. The crawling-based data collection mechanism of LEADS differs from the traditional Web crawling by performing the crawling tasks in geographically distributed micro-clouds rather than in a single cloud, which aims at increasing the efficiency of data collection and the scalability of the system. Therefore, the crawlers located in different micro-clouds may need to communicate with each other to determine the set of web pages they should crawl to ensure the overall performance. We present an early implementation of the distributed crawlers, as well as the distribution of URLs among crawlers in Section 6.

The user-aided data collection mechanism of LEADS allows users to publish, in near real-time, the web pages that they are visiting or creating, to the LEADS platform through the user-publishing interface. Users can also publish metadata like tags, reviews, etc. to pages, providing additional information that can be leveraged for data analysis served by LEADS. Data will be published to only one micro-cloud and then stored to other micro-clouds if necessary, according to the data placement policy of the LEADS Key-Value store. The user-aided data collection mechanism is an enhancement of the crawling-based data collection mechanism.

3. Specification of user-aided data collection

3.1 High-level design

The user-aided data collection mechanism relies on end users to discover data that are difficult to be gathered by the crawling-based data collection mechanism. As we have discussed, such data include the data in the Web that are often present in dynamically generated web pages, and user generated metadata in the form of tags, comments, etc. to provide complementary information on the web data. To discover and collect data from users, the user-aided data collection mechanism allows users to explicitly or implicitly publish their data through a user publishing interface. These data will be then selectively fetched and updated in the LEADS storage layer.

Figure 2 depicts the high level architecture of the user-aided data collection framework of LEADS, focusing on a single micro-cloud. When a user accesses the data in the Internet, she can decide whether to associate any metadata with this URL that she is visiting. Then the URL (and its metadata) will be sent to the user-aided data collection system of a micro-cloud. The specific micro-cloud to which the URL (and its metadata) should be sent can be determined with respect to different requirements on the system. In LEADS, the selection of the micro-cloud in charge of the URL (and the metadata) depends on the key assignment techniques used by WP2 and WP4. Besides, users of LEADS can also create their own data and create a URL to them, and explicitly report the URL and its metadata to the user-aided data collection system of a micro-cloud. We present the design and the implementation of the user-publishing interface in Section 4.

All the user published data are sent to the pre-processing unit of the micro-cloud, which is in charge of filtering and merging the received data before putting them to the data collection. Data that successfully pass pre-processing is then inserted in the Key-Value store. If the content in a web page is published, the URL of the web page, as well as the associated data and metadata will be inserted in the Key-Value store. Otherwise, if the content of a web page is not published, the URL of the web

page will be added to the frontier[1] of the user-aided crawler to fetch the content pointed by the URL. Note that the pre-processing will be addressed in the following deliverable. In this deliverable, we assume that all the pages and the associated metadata published by users are valid, i.e., the pre-processing unit is ignored in the current data flow. We evaluate, based on this assumption, to which extent the user-aided data collection allows to enlarge the data collection obtained by crawling-based data collection in Section 5. This gives an upper bound on the ability of user-aided data collection to gather data for the LEADS platform.

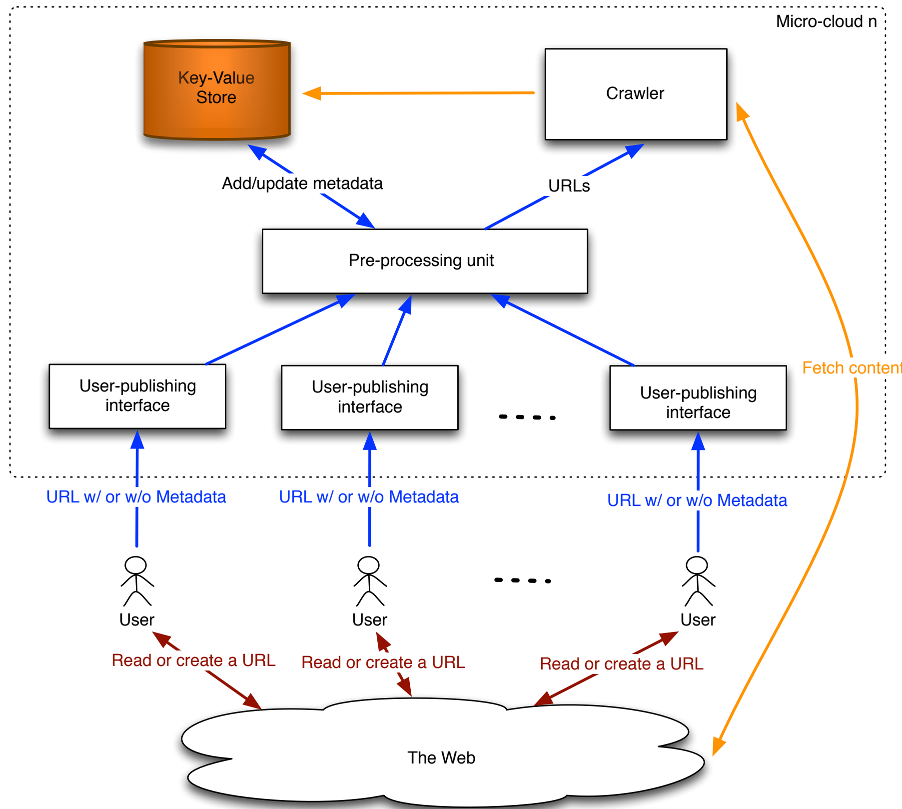


Figure 2: Design of user-aided data collection framework in a micro-cloud

3.2 Requirements

The objective of user-aided data collection is to efficiently discover data that is difficult to be discovered by the crawling-based data collection, and to effectively enrich the data (collected either by the crawlers or provided by users) with user-generated metadata. The data collected user-aided can be then merged to the data collected by crawlers to build a Web graph, which is then used by LEADS for its Web graph service. To this end, the user-aided data collection mechanism needs to meet the following requirements:

- High quality data. The quality of data is key to the quality of the data services provided by LEADS, while users may publish different kinds of data or metadata with various qualities for different purposes. It is important to filter out low quality and malicious data to make sure only high quality data are collected.

[1] Frontier is the list of URLs to be crawled by a crawler.

- Efficient crawling. The user-aided data collection should not fetch all the data published by users, but should focus on the data that are difficult to be collected by the crawling-based data collection, unless the data have high requirement on its freshness and need to be updated in near real-time. This allows avoiding redundant fetching of the same data and ensuring efficient crawling.

3.2.1 Requirements on quality

3.2.1.1 Crawl ordering

Web crawlers usually order the URLs to crawl according to their estimated importance values, such as the quality of their content [CGP98], their connectivity with other pages [EMT04] and their impact on the associated applications [PO05]. Different from crawling, the user-aided data collection mechanism of LEADS mainly focuses on collecting the content that are generated by users. Clearly, if users publish data to LEADS, they probably want to see that data available for certain data services as soon as possible. Therefore, the user-aided data collection needs to ensure near real-time crawling of data that is published by users. The freshness of data is a quality feature to be taken into account when designing the crawling order for the user-aided data collection system.

3.2.1.2 Dealing with malicious publishing

Users of LEADS are given the freedom to publish their own data as well as associate metadata and attach it to data that is already stored in the LEADS platform. However, users may abuse the system by intentionally publishing irrelevant or even malicious content for various purposes. For instance, users may associate fake reviews to some products to bias or even damage the reputation of those products or promote their own products [LNJ+10]. In fact, malicious data can take many forms, possibly manifesting as a web page, an annotation, a user profile, or an automated review. The motivations of such data can be advertising, self-promotion, disruption, curiosity, or disparaging a competitor [KGM07]. To ensure the quality of the data services provided by LEADS, especially those related to sentiment mining, it is important to keep the data collection clean by detecting and eliminating malicious publishing.

3.2.2 Requirements on efficiency

3.2.2.1 Reducing redundancy

The user-aided data collection system of LEADS allows users to publish their own data either explicitly or implicitly to the system. Unselectively collect all data whenever it is published by users would cause unnecessary bandwidth consumption. This is because different users may publish the same data at different time across the network. For instance, users may visit the same web page describing an emerging event. Users may also generate same tags for annotating some content in the Web. In such cases, unless new data is available, we do not want to fetch the same content several times simply because they are independently published by different users at different time. Moreover, users may publish incremental updates, such as tags, comments, etc. on the same page. Even if crawling the page each time brings new data, this may still be a waste of bandwidth as only increments are necessary to update the data collection. Besides, users may publish content that are already available in LEADS through the crawling-based data collection. Therefore, it is important for the user-aided data collection system to pre-process the data published by users, before putting them to the frontier of the user-aided crawler, to avoid redundant data fetching as much as possible.

3.2.2.2 Crawler throughput

User published content will be assigned to a user-aided crawler for fetching. Similar to a web crawler, a user-aided crawler also runs multiple processes in parallel [CG03, HN99] to increase throughput until the available bandwidth for fetching the data saturates. Once a user-aided crawler saturates its

bandwidth or there is too much data to be processed before passing it to the crawler, a decision should be made to shift the newly published content to other crawlers with available resources to ensure the overall throughput of the system.

In the next section, we present the design and implementation of the user-publishing interface, as well as how the user-aided data collection system meets the requirements on crawl ordering and reducing redundancy.

4. Design and implementation of the user-aided publishing interface

4.1 Functionality of the user-aided publishing interface

The user-aided data collection mechanism relies on users to provide data to the platform. Specifically, users can send the web pages they create or they visit, along with their own tags or comments to the Key-Value store of LEADS. As we have seen, a key component for enabling this functionality is what we call the user-aided publishing interface, which allows users to either explicitly or implicitly send their data to LEADS. Basically, the publishing interface will perform the following actions:

- Allow users to explicitly publish the data they create, typically linked by a URL, to the platform. The URL of the published data is explicitly provided by user. Users can associate metadata, in the form of tags, comments, etc., to provide additional descriptions of the published data. The metadata can be empty if users do not have any metadata to associate with the data they publish.
- Allow users to implicitly publish the web pages they visit to LEADS. The URL and the corresponding data are automatically extracted from the web pages being visited, without any additional input from users. Users can associate metadata to the pages they are visiting. The metadata can be empty if users do not have any metadata to associate with the web pages they are visiting.

The user publishing interface will report the published data in the form of <URL, timestamp, data> triples to the user-aided data collection system. The URL is the link to the published data and is thus mandatory. The timestamp is the Unix timestamp that indicates when the data is published. Data is in JSON format, composed of a set of fields, such as the identifier of the user who makes the publication, the title of published data, the metadata associated to the published data.

4.2 Design of the user-aided publishing interface

The user-aided publishing interface is conceived as a plug-in to a Web browser. Users would be able to use it easily on their end devices. This plug-in provides the above described functionalities and the published data will be sent to LEADS to serve the potential data mining tasks.

Figure 3 depicts the user-aided publishing interface. In order to publish data, user first needs to go to the web browser. By clicking on the LEADS icon on the top-right corner, the publishing interface appears as a popup window. The user-aided publishing interface supports two types of publishing: explicit publishing and passive publishing.

Figure 3: User-aided publishing interface of LEADS

Figure 4 further depicts the important fields in the publishing interface.

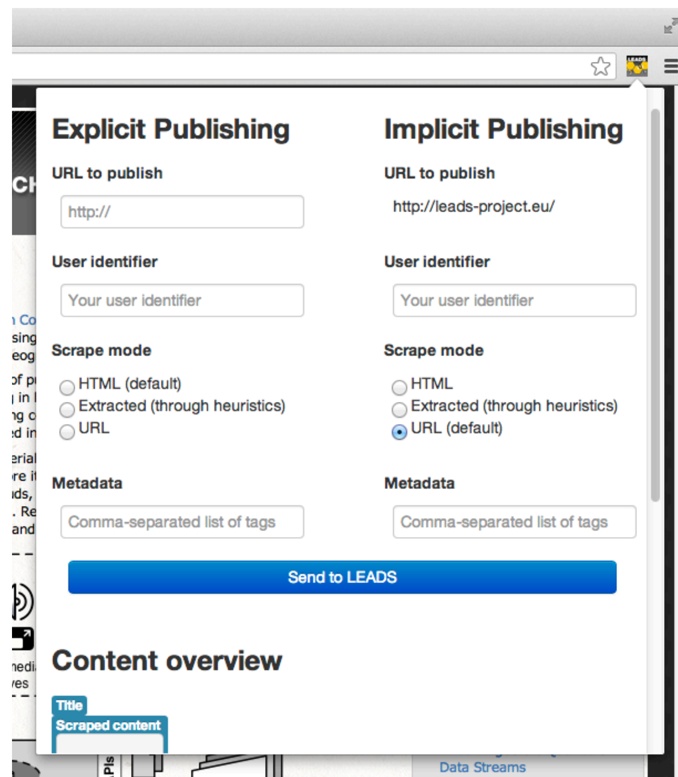


Figure 4: Popup publishing window

For the explicit publishing, a user needs to type or copy the URL that links to the data she wants to publish in the required field "URL to publish". Then, user can further choose whether she wants to publish only the URL or the data linked by the URL to LEADS, by selecting from the list of three scrape modes:

- HTML: The raw HTML file linked by the user provided URL will be sent to LEADS as the published data. Figure 5(a) gives an example of the raw HTML file.
- Extracted: The main body of the HTML file linked by the user provided URL is first extracted according to a heuristic that uses a variety of metrics, such as content score, class names, element types, to find the content that is most likely to be the content a user would like to read in the HTML file. Figure 5(b) gives an example of the extracted content from the same web page. Then the extracted content will be sent to the LEADS platform as the pub-

lished data.

- URL: Only the URL will be sent to LEADS as the published data. The data itself would be later fetched by the user-aided crawler if it is not already available in the Key-Value store of LEADS.

Scraped content

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
 version="XHTML+RDFa 1.0" dir="ltr" xmlns:content="http:
//purl.org/rss/1.0/modules/content/" xmlns:dc="http://pu
rl.org/dc/terms/" xmlns:foaf="http://xmlns.com/foaf/0.1/
" xmlns:og="http://ogp.me/ns#" xmlns:rdfs="http://www.w3
.org/2000/01/rdf-schema#" xmlns:sioc="http://rdfs.org/si
oc/ns#" xmlns:sioct="http://rdfs.org/sioc/types#" xmlns:
skos="http://www.w3.org/2004/02/skos/core#" xmlns:xsd="h
ttp://www.w3.org/2001/XMLSchema#" class="js"><head profi
le="http://www.w3.org/1999/xhtml/vocab">
```

(a) Scrape mode: HTML

Scraped content

```
LEADS is a new European integrated project funded in par
t by the European Commission under the FP7. It groups to
gether 3 universities and 4 companies with the common go
al of building and showcasing a novel Cloud service mode
l named Data-as-a-Service, atop an innovative infrastruc
ture based on an elastic collection of geographically di
stributed micro-clouds.
```

(b) Scrape mode: Extracted

(c) Scrape mode: URL

Figure 5: Example of different scrape modes

The default scrape mode is HTML for explicit publishing. The underlying motivation is that once users create new web pages and publish them to the platform, they would expect the published data to be available for their (or others') data processing tasks as soon as they perform the publishing operations. Users are also given the freedom to publish or simply check the main body of the created web pages by clicking on the scrape mode "Extracted" to make sure that the page contains the right data to publish. Besides, it is also possible for users to only publish the URLs to their data. If users are editing their data or they will edit their data on the URLs soon, they can simply send the URLs to their data to register the publication and the content can be fetched later to the Key-Value store of LEADS by the user-aided crawler. After selecting the scrape mode, users can attach metadata like tags to describe the data they are publishing by taping a set of comma-separated keywords into the field "Metadata".

Besides, users can decide if they would like to publish the data anonymously or not. In case that they may need to retrieval the data, especially the metadata that are only published by them, they can associate their user identifier to their publication. This can be achieved by simply typing a name in the field "User identifier". Note that the current publishing interface does not support user authentication but allows users to use any identifier they like for their publications. Authentication mechanism can be later integrated to the interface depending on the specific needs of the project. On the other hand, users can publish their data anonymously by leaving the field "User identifier" blank.

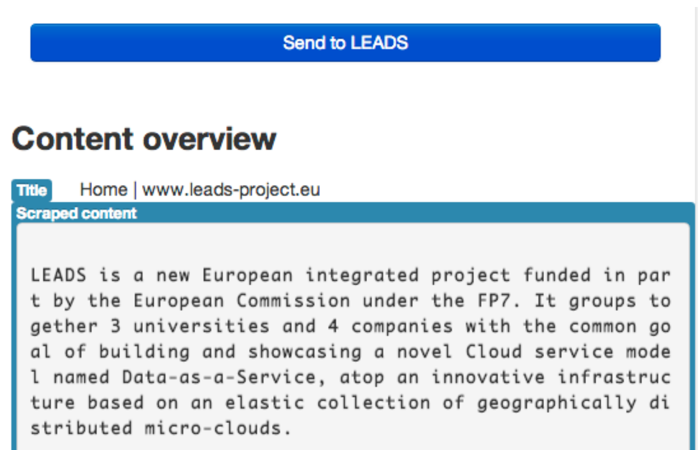


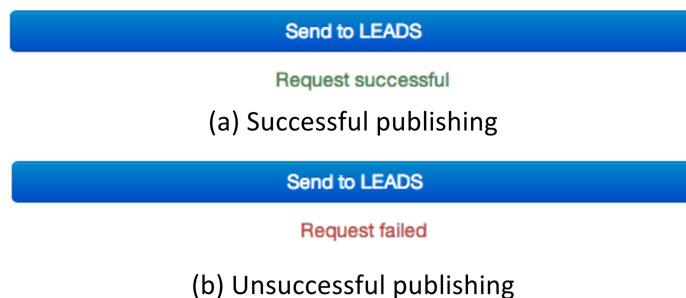
Figure 6: Example of "Content overview" field

Before pressing the button "Send to LEADS" to publish the data, users can review the content of the data they are going to submit in the region below "Content overview", as shown in Figure 6 to make sure only the desired data would be submitted.

For the implicit publishing, when a user is visiting a web page, the URL to that page will be automatically extracted. The user does not need to specify the URLs to the data they would like to publish as for the explicit publishing. There are also three scrape modes, namely HTML, Extracted, and URL, to obtain and publish the content of the page that is visiting by the publishing user. Different from the explicit publishing, the default scrape mode is set to URL for the implicit publishing. This is based on the observation that once a user publishes data around a web page that already exists in the Web, it is possible that the crawler of LEADS, which continuously fetches the web pages to enlarge its data collection, has already known this page. Unless the page is updated, sending the content of the page is not specifically useful for enlarging the data collection but only consumes additional resources for transmitting it. However, users are also given the possibilities of publishing the full HTML or the extracted content to LEADS as for the explicit publishing. These options are useful for at least two situations. First, if the user who is visiting the page finds the content very interesting and recent, she may decide to publish the content of the page directly so that others may have an opportunity to see the same content earlier. As we will see in Section 5.2, since it takes time for the user-aided crawlers to reach certain pages according to their policies of ordering the URLs in its frontier, certain pages can indeed published earlier by the users who visit them than by the crawlers. Second, if the user is visiting (and editing) a page that she created before, it is natural that the user would like to publish the content along with its URL to make it available as soon as possible in the platform.

Similarly, users can provide metadata, like tags, comments, etc., in the corresponding field as for the explicit publishing, and decide whether to publish the data anonymously or through their preferred user identifiers.

Finally, for both explicit and implicit publishing, by pressing the button "Send to LEADS", the publishing process is completed, and a message will show below the button, to indicate whether the data is successfully published to LEADS, as shown in Figure 7.



(a) Successful publishing

Send to LEADS

Request failed

(b) Unsuccessful publishing

Figure 7: Status of user-aided publishing

4.3 Implementation of the user-aided publishing interface

4.3.1 Overview

The on-publishing interface is implemented as an extension to the Chrome web browser[2]. An extension is a zipped bundle of files—HTML, CSS, JavaScript, images, etc. that add functionality to the Google Chrome browser. Extensions allow adding functionality to the Google Chrome browser without the necessity of diving deeply into its native code. Extensions are essentially web pages. They can use all the APIs that the browser provides to web pages to interact with web pages or backend servers using content scripts or cross-origin XMLHttpRequests. Extensions can also interact programmatically with browser features and support exchanging messages between an extension and its content scripts or between extensions.

Like most of the Chrome extensions, the user-aided publishing interface requires adding a user interface to the Google Chrome browser in the form of a browser action. This browser action is used to put an icon of LEADS, as we have seen in Figure 3 in the main Google Chrome toolbar, to the right of the address bar. Once clicking on the icon, a popup window will appear to allow users to publish their data.

The extension of the user-aided publishing interface includes the following important files:

- A manifest file: This file is called *manifest.json*. It specifies the information about the extension, such as what are the most important files and what are the capabilities that the extension will have.
- Two HTML files: These files are ordinary HTML pages that display the extension's user interface. The two HTML files for the user-aided publishing interface are as follows:
    - An options page, called *options.html*, which defines the layout of the page for setting the options of the extension.
    - A popup page, called *popup.html*, which defines the layout of the popup window when the user clicks on the icon of LEADS to publish her data.
- Four JavaScript files: These files define the actual operations the extension needs to perform according to the definition in the HTML files. The three JavaScript files for the user-aided publishing interface are as follows:
    - A script related to the options page, called *options.js*, which specifies the server backend that would receive the data published by user through the extension, as well as its interaction with the extension.

[2] Chrome extension. http://developer.chrome.com/extensions/extension.html

- A script related to the popup page, called *popup.js*, which defines how to capture user interaction with the popup page to publish the data.
- Two scripts, called readability.js and leads.js, which define how to execute the context of a web page that has been loaded into the browser (referred to as *content script*), such as extracting the title and the content of a web page as we have seen in Figure 6.

We explain in the next section how the important components of the user-aided publishing interface are implemented.

4.3.2 Implementation of key components

The first thing to implement the publishing interface is to create is a manifest file with the name "manifest.json". This file a JSON-formatted table of contents, containing properties like the name of the extension, i.e., LEADS user-aided publishing interface, its description, its version number, and its communication channel with the server backend, i.e., options.html, which we explain shortly. Basically, this file is to declare to the Google Chrome browser what the extension is supposed to do, and what permissions it requires in order to do them. Figure 8 gives a snapshot of the "manifest.json" file we use for defining the publishing interface. It worth noticing that in terms of permissions, only the content of "http" pages will be extracted and sent to LEADS, while "https" are ignored for privacy considerations. As mentioned before, this file also specifies that the format of the popup window is defined by popup.html, and the main JavaScript files that define the interaction with the page that is loaded to the browser (i.e., being visited by the user who would like to publish data) are readability.js and leads.js which we also explain shortly.

```
{
  "manifest_version": 2,

  "name": "LEADS user-aided publishing interface",

  "description": "Extension for publishing web pages to the LEADS infrastructure",
  "version": "1.0",

  "options_page": "options.html",

  "permissions": [
          "activeTab",
          "tabs",
          "http://*/*"
  ],

  "browser_action": {
          "default_icon": "icon-leads.png",
          "default_title": "Make this page red",
          "default_popup": "popup.html"
  },

  "content_scripts": [{
          "matches": ["http://*/*"],
          "run_at": "document_idle",
          "js":["js/readability.js", "leads.js"],
          "all_frames": false
  }]
}
```

Figure 8: Manifest.json for the user-aided publishing interface

We also implement an options page to customize the behavior of the user-aided publishing interface. This is achieved through the files “options.html” and “options.js”. In the current version, we simply specify through the options the backend server where the data will be published to and the pass-word that identifies the LEADS users. The file options.html defines the page what would be open at a new tab when user edits the “Options” button in the extension management page at “chrome://extensions”. The options page is shown in Figure 9.

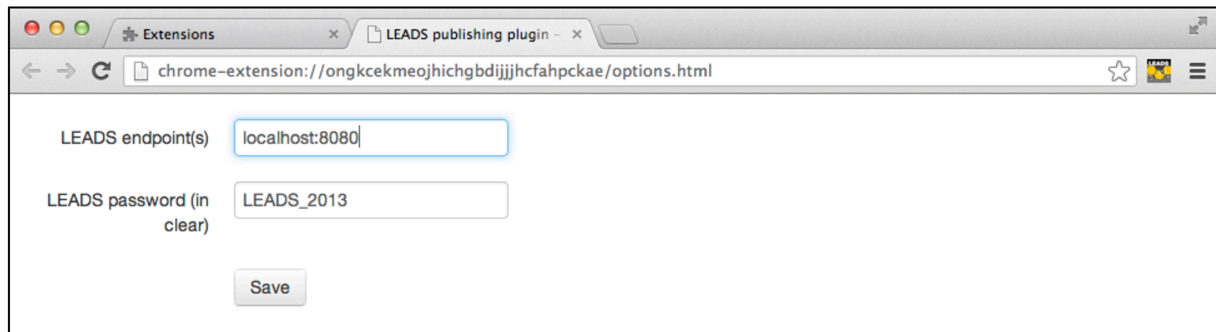


Figure 9: Options page of the user-aided publishing interface

The field “LEADS endpoint(s)” requires the name of the server backend. As the pre-processing unit of the user-aided data collection will be later incorporated to LEADS in the following deliverables, the published data will only be sent to a temporary Key-Value store, which resides in the web server that will perform the pre-processing tasks. How the options page receives the endpoints and password typed by LEADS users, and configures the setting of the communication channel with the backend is defined in the JavaScript file “options.js”. This is achieved by implementing the following functions shown in Figure 10. The function restore_options() loads the default values of endpoint and pass-word using the LocalStorage API[3] and the function save_options() captures the values of endpoint and password provided by users and overwrites their default values using the same API.

```
function restore_options() { … };

function save_options() { … };

document.addEventListener('DOMContentLoaded', restore_options);

document.querySelector('#save').addEventListener('click', save_options);
```

Figure 10: Skeleton of options.js for the user-aided publishing interface

The key popup window, which is shown when the user clicks on the LEADS icon, as well as the related actions, which user performed inside the popup to publish data, are implemented through the files “popup.html” and “popup.js”. The layout of the popup as we have seen in Figure 4 is specified in “popup.html”, which defines two types of input from users:

- Type “text”, which allows users to type the user identifier, the URL to the data to publish in case of explicit publishing, and the metadata.

[3] LocalStorage API. https://developer.mozilla.org/en-US/docs/Web/Guide/DOM/Storage?\redirect\-locale=en-US&redirectslug=DOM%2FStorage

- Type “radio”, which allows users to choose from the list of scrape modes which content related to a URL they would like to send to the Key-Value store of LEADS.

Besides, it contains the region “Content overview” to show the preview the data before sending them to the backend, and the button “Send to LEADS” that actually sends the data.

The specific actions, including interaction with users and communication with server backend are defined in the file “popup.js”. Figure 11 lists the key functions implemented in the file.

addEventListener() monitors the user actions in the popup window. Once a “click” action is captured, the function send() is called for the selected scrape mode, and the function publish() is called when the button “send to LEADS” is pressed. The function send() emits an event about the scrape mode, which is captured by the listener defined in the file “leads.js”. Figure 12 depicts the skeleton of the listener.

The listener reads the content of the web page being visited by the user in the browser, and extracts the HTML file (in the mode “HTML”), the main body of the HTML (in the mode “Extracted”) from that web page or nothing (in the mode “URL”), as well as the title of the page. In case that the main body of the HTML needs to be extracted, the function grabArticle() defined in the file “readability.js” is called to perform the content extraction. These values are then assigned to the JSON object that is to be sent to the Key-Value store of LEADS. The function publish() captures the user identifier, the list of metadata provided by the user and assign their values to the same JSON object, and then send the <key,value> pair with the URL as key and the JSON object as value to the server backend by calling the function XMLHttpRequest.send().

Regular web pages can use the XMLHttpRequest object to send and receive data from remote servers, but they are limited by the same origin policy. An extension can talk to remote servers outside of its origin by setting up the cross origin permissions. This is achieved by adding host match patterns (i.e., http://*/*) to the permission section of the “manifest.json” file as shown in Figure 8. This match pattern allows HTTP access to all reachable domains. Note that only non-secure HTTP access is only granted to the publishing interface by this match pattern, and secure HTTP access is not granted for privacy concerns in the context of LEADS.

The function sendSuccessUpdate() and the function sendFailedUpdate() are called inside the function publish() after sending the <key, value> pair to the server backend to indicate whether the publication of data is successfully accomplished. These functions are also in charge of showing the status message that we have seen in Figure 7.

```
document.addEventListener('DOMContentLoaded', function() {

        document.querySelector('#ImplicitHtmlRadio').addEventListener('click',
                function() {
                        send('implicitHtml');
                });
        document.querySelector('#ImplicitExtractedArticleRadio').addEventListener('click',
                function() {
                        send('implicitExtractedText');
                });
        document.querySelector('#ImplicitUrlRadio').addEventListener('click',
                function() {
                        send('implicitURL');
                });
        document.querySelector('#ExplicitHtmlRadio').addEventListener('click',
                function() {
```

```
                    send('explicitHtml');
            });
    document.querySelector('#ExplicitExtractedArticleRadio').addEventListener('click',
            function() {
                    send('explicitExtractedText');
            });
    document.querySelector('#ExplicitUrlRadio').addEventListener('click',
            function() {
                    send('explicitURL');
            });

    document.querySelector('#publish').addEventListener('click', publish);

  });

  function send(contentKind) { … };

  function publish() { … };

  XMLHttpRequest.send() { … };

  function sentSuccessUpdate() { … };

  function sentFailedUpdate() { … };
```

Figure 11: Skeleton of popup.js for the user-aided publishing interface

```
chrome.runtime.onConnect.addListener(function(port) {

  port.onMessage.addListener(function(msg) {
          if (msg.query == 'getContent') {
                  switch (msg.kind) {
                  case 'implicitHtml':
                          port.postMessage({ … });
                          break;
                  case 'implicitExtractedText':
                          port.postMessage({ … });
                          break;
                  case 'implicitUrl':
                          port.postMessage({ … });
                          break;
                  case 'explicitUrl':
                          port.postMessage({ … });
                          break;
                  case 'explicitExtractedText':
                          port.postMessage({ … });
                          break;
                  case 'explicitUrl':
                          port.postMessage({ … });
                          break;
          }
  });

});
```

Figure 12: Skeleton of lead.js for the user-aided publishing interface

4.4 Interaction with the backend

As we have mentioned, the data published to LEADS is sent to the Key-Value store through a JSON object that includes both the data and the metadata. The JSON object is identified by its key, which is the URL that links to the page that contains the published data. Figure 13 gives the format of the JSON object that is sent to LEADS through an example.

```
{
  url: http://leads-project.eu/
  title: Home | www.leads-project.eu
  userId: BM-Y!
  content: LEADS is a new European integrated project funded in part by the European
           Commission under the FP7….
  tags: {
         eu project,
         fp7,
         homepage
  }
  timestamp: 1378720549635
}
```


Figure 13: Format of JSON object published to LEADS

Data will be published to the Key-Value store of LEADS through its PUT API. Since the user published data will be processed before being sent to the Key-Value store of LEADS, we also implement a temporary cache to host them in the web server that will perform data pre-processing.

To this end, we define the temporary cache as in Figure 14. The class BasicLeadsAuthenticator relies on the password that users provide in the options page (Figure 9) to grant users the right to publish (write) data to the Key-Value store. The class PublicationResource implements the publish and read operations to the Key-Value store using its PUT and GET API.

```
public class PublicationService extends Service<PublicationConfiguration> {

     @Override
    public void run(PublicationConfiguration configuration, Environment environment) throws
             Exception {
    String dbString = configuration.getDbString();
    Cache cache = (Cache) Class.forName(configuration.getDbClass()).newInstance();
    environment.addResource(new PublicationResource(cache.connect(dbString)));
    environment.addHealthCheck(new DBHealthCheck(dbString, configuration.getDbClass()));
    environment.addProvider(new BasicAuthProvider<LeadsUser>(new BasicLeadsAuthenticator(),
             BasicLeadsAuthenticator.REALM));
    }

    public static void main(String[] args) throws Exception {
      new PublicationService().run(args);
    }
}
```

Figure 14: Implementation of the backend Key-Value store

4.5 Deployment of the user-aided publishing interface

To install the user-aided publishing interface, a LEADS user simply needs to follow the steps listed below:

- Visit “chrome://extensions” in the Google Chrome browser.
- Select the “Developer mode” checkbox in the top right-hand corner of the loaded page.
- Click the “Load unpacked extension…” button to pop up a file-selection dialog.
- Navigate to the directory of the extension files and select it.

Once these steps are completed, the publishing interface is successfully loaded. In order to use it to publish data, user first need to set up the options defined by the options files. To this end, the user needs to click on the button "Options" on the page "chrome://extensions" and then provide the required information on the appearing page as shown in Figure 9. The field "LEADS endpoint(s)" requires the name of the server backend. User can specify the server located in micro-cloud that is geographically close. The name of the micro-cloud to where the data will be sent can be obtained by an external service that computes the distance of the user to micro-clouds through her IP address. Only authenticated users knowing "LEADS password" can send their data to LEADS through this interface. After saving the backend information, the publishing interface is ready to use.

5. Evaluation of user-aided data collection

We evaluate in this section the ability of the user-aided data collection mechanism to gather data in addition to the crawling-based data collection mechanism. To this end, we compare the data that can be gathered through user-aided publishing to what can be gathered through data crawling. This gives us an estimation on the benefit of using user-aided data collection for gathering new and high quality data in addition to the crawling-based data collection.

5.1 Experimental setup

The objective of this experiment is to compare the data gathered by the user-aided data collection mechanism to that gathered by the crawling-based data collection mechanism. We use a snapshot of the web crawl of Yahoo Search engine to approximate the amount of data that can be gathered by the crawlers of LEADS. This is a very large crawl in the order of billions of web pages, while for confidentiality reasons; we cannot disclose the exact size of this dataset. As we have described, users of LEADS can publish data either in an explicit or an implicit way. In this experiment, we focus on implicitly published data, which is collected from a browser log available at Yahoo. This log contains for every registered user, the web pages that are visited by the user. We sample the log for the first day of each month during half a year, where each day consists of the web pages visited by millions of users. Note that in LEADS, users only selectively publish the web pages they visit, while this log records all the pages they visit. Therefore, the experiment on this log gives an upper bound on the impact of the implicit publishing. Evaluating large-scale explicitly published data is impractical given the huge human efforts it will require in the data-collecting phase. We exclude this part from the evaluation. Our experiment is conducted on a large Hadoop cluster.

5.2 Experimental results

As we have presented, publishing in LEADS implies publishing both the URL (and content linked by the URL) and the metadata personalized to each user. Clearly, metadata collected by the user-aided publishing interface is a net gain to LEADS as such data is not supposed to be collected by the crawlers. We thus focus on evaluating the URLs published by users.

We first evaluate how many URLs visited by the users cannot be crawled by the crawlers that solely gather data by following the links. We achieve this by querying every URL that exists in the browser log against the web crawl to see if it also exists in the latter. If a URL does not exist, it means this URL, as well as the data linked by this URL, are not reachable by the crawlers[4] and it is thus important if they can be gathered through user-aided publishing of LEADS. Therefore, the larger the number of

[4] It is also possible that the crawler has not reached this page, but will reach it later. Yet, this means gathering the corresponding data through user-aided publishing is still beneficial than gathering it through crawling.

such URLs, the higher the benefit of the user-aided data collection. Figure 15 shows the percentage of the URLs that only exist in the browser log w.r.t. all the URLs that exist in the same day for the six sampled day.

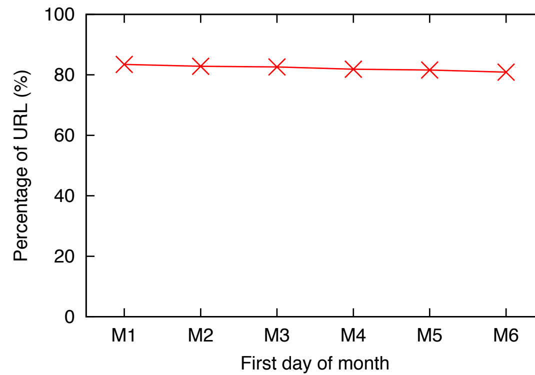


Figure 15: Percentage of URLs that cannot be reached by crawling-based data collection

We observe that more 80% of URLs that are visited by user on each day were not crawled by the crawlers. The slight decrease of the percentage with time is because we use a fixed web crawl in the experiment and more pages are crawled by the crawlers in each month.

To explain the high number of visited URLs that were not collected by the crawlers, we manually inspect the content of a sample of 1000 pages. We find that half of the pages visited by users are private pages like Facebook and emails that require users to log in to view their content, and only half of them contain meaningful content and have open access to all users. Therefore, in the context of LEADS, at most 40% of pages visited by users in each day are complementary to the data gathered by the crawlers. Remind that this is only an upper bound of the amount of pages as users of LEADS have the freedom to selectively publish the pages they visit and do not need to publish all of them.

For the URLs that are both crawled by the crawlers and visited by users, we compare the timestamps when they are crawled to the timestamps when they are visited to see if users have the potential to publish valuable data before they are collected by the crawlers. Figure 16 shows the percentage of URLs having smaller timestamp in the browser log than in the web crawl w.r.t. all the URLs that exist in the same day for the six sampled day. We observe that from 1.4% to 2.3% of URLs visited by users on a day are collected by the crawlers by up to 6 months later. This confirms that user-aided publishing provides LEADS a good opportunity to collect data earlier.

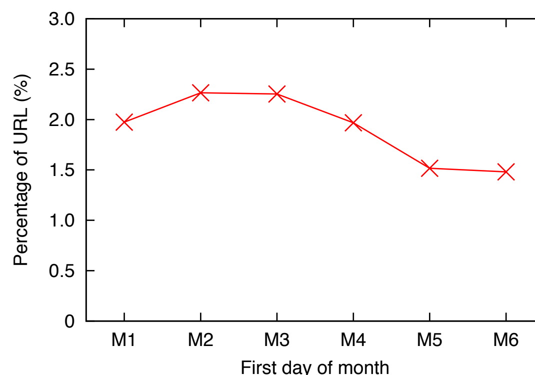


Figure 16: Percentage of URLs that can be published earlier by users

Finally, to understand why certain pages are more likely to be visited by users rather than be reached by the crawlers, we compute the percentage of static URLs and dynamic URLs using regular expression on the URL patterns. We observe that more than 70% of pages that are visited by users but are not collected by crawlers are dynamic. In other words, the content of these web pages are generated by querying a backend database with the parameters filled in the URL paths. This confirms our motivation of user user-aided publishing to collect data, as the links to the dynamic pages do not exist in prior, and it is also difficult for the crawlers to automatically generate the parameters in the URL paths to fetch their content. This is particularly interesting for forums that are public but require a dynamic URL.

6. Design and implementation of the web crawler

We present in this section the design and implementation of the crawler. Although the focus of this deliverable is the user-aided publishing mechanism, we decide to provide a specification of the crawling-based data collection, and work in advance on the subject in order to ease early integration and testing with the other work packages. The crawler that we present in this section is a basic prototype. Further enhancement of functionality and performance will be addressed in the following deliverables.

6.1 Specification of crawling-based data collection

The crawling-based data collection mechanism of LEADS relies on geographically distributed crawlers to discover and fetch data from the Web. Each crawler is only in charge of crawling a subset of URLs. Therefore, it is important to assign URLs to the crawlers that are the most appropriate to fetch their content, to ensure both efficient and cost-effective crawling. Instead of assigning the URLs statistically [CPJ+08] to the distributed crawlers, the crawling-based data collection mechanism of LEADS employs an user-aided approach: a URL is dynamically assigned to a crawler once it is discovered for the first time. With this approach, workload in different crawlers can be adaptively adjusted to optimize the crawling efficiency and the operational cost of the crawlers.

Figure 17 illustrates the high-level architecture of the crawling-based data collection system of LEADS. Each crawler starts crawling with a pre-selected seed set of URLs. Once a crawler fetches a page, the crawler parses the page to extract new URLs to fetch. Instead of directly adding the new URLs to the local frontier, the crawler determines, for each new URL, which is the most appropriate crawler to fetch its content. Based on this decision, the crawler periodically exchanges a list of new URLs with other crawlers, and keeps some URLs for crawling locally. The lists of URLs received from other crawlers are added to the local frontier for further crawling.

The objective of the crawling-based data collection is to efficiently fetch the data that are publicly available in the Internet by following the link structure of pages. The collected data can be then leveraged to build a Web graph, which is then used by LEADS for its Web graph service. To this end, the crawling-based data collection mechanism needs to meet the following requirements:

- <u>High quality data.</u> The crawlers should fetch the data with high quality with high priority, given the huge amount of data that are publicly available. Moreover, for data that change frequently, it is important to maintain their freshness once available in the LEADS platform. The quality of the data directly determines the quality of the data services, e.g., the Web graph service, provided by LEADS.

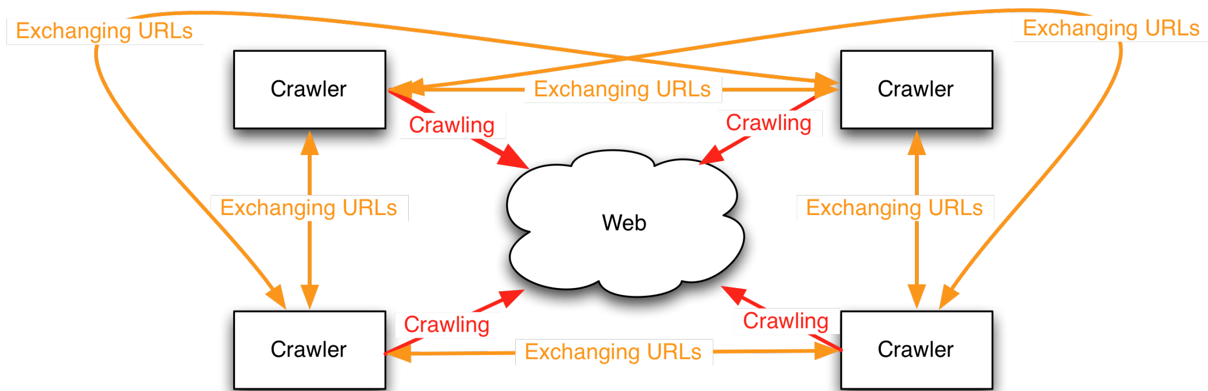


Figure 17: Architecture of the crawling-based data collection framework

- Efficient crawling. The crawlers should fetch the data efficiently. Given the distributed nature of the crawling-based data collection system of LEADS, it is important to well partition the crawling tasks among crawlers, so as to maximize the overall throughput for fetching the data.
- Cost-aware crawling. Distributing the crawlers in geographically distributed locations provides an opportunity to reduce the overall cost of crawling by leveraging the differences in cost among crawlers. The crawling-based data collection system of LEADS should take this feature into account to achieve cost-aware crawling.

Note that the requirements on quality are basic requirements that all the web crawlers should meet, the requirements on efficiency are tailored to geographically distributed setting where the LEADS platform is built on, and the requirements on cost are specific to LEADS that make the crawlers of LEADS different from the existing distributed web crawlers. Besides, the crawlers of LEADS need to be as polite as possible to the web servers that hold the pages to crawl, as all the web crawlers.

6.1.1 Requirements on quality

6.1.1.1 Crawl ordering

The Internet contains many billions of Web pages. Major search engines like Google, Yahoo! and Bing have indexed more than 50 billion Web pages until 2013[5]. While the size of the Internet keeps growing, it is impossible for the crawlers to fetch all the publicly available data. Therefore, the crawlers need to determine which pages should be crawled first, in order to ensure the most important data are available for the desired service. Generally, a crawler only needs to download pages just a few levels, no more than 3 to 5 "clicks" away from the start page, to reach 90% of the pages that users actually visit [BC05]. A notable technique to improve the download content quality is to fetch the links in the frontier in decreasing order of the predicted importance values [CGP98, NW01]. For instance, search engines often relies on quality-based metrics, such as linkage among pages, content of a page, to prioritize URLs with higher importance in crawler's frontier [EMT04]. More advanced techniques include prioritizing URLs from the sites that contributed more to the search results [SO05].

[5] The size of the World Wide Web (The Internet). http://www.worldwidewebsize.com/.

Besides Web search that aims at collecting data from all the index-able Web, focused crawlers prioritize URLs specified to given applications like social networking sites for crawling first [BOM12, CDF+11]. Similarly, the crawling-based data collection system of LEADS cannot fetch all the data in the Internet. It needs to incorporate appropriate crawling order mechanism to make sure that the most important data with respect to the services LEADS provides are efficiently crawled.

6.1.1.2 Re-crawling

As the Web evolves, the content of certain pages keep changing. Therefore, it is important to maintain the freshness of the data stored in the LEADS platform. Keeping the available data fresh requires re-fetching already fetched pages. A number of studies investigated the evolutionary properties of the Web [ATT+09, FMN+04, NCO04] and proposed page refreshing policies for crawlers. Some policies aimed to refresh the content that is more important more often [CG00] or more likely to change [CG03, EMT01]. Others tried to maximize the impact on search result quality [FCV09, PO05]. As LEADS does not focus on search, the crawling-based data collection system of LEADS will be based on the importance of the pages to the data services LEADS provides and the likelihood of changes to determine the pages to re-crawl.

6.1.2 Requirements on efficiency

6.1.2.1 Crawler throughput

Given the huge amount of data available in the Internet, to build a large data collection, it is very important to fetch the data efficiently. Throughput is a measure of the amount of Web content fetched by the crawler per unit of time. Obviously, high throughput is desirable. This is because it ensures either larger size of the data collection or fresher content in the data collection. A crawler typically runs multiple processes in parallel [CG02, HN99] to increase throughput as long as the available bandwidth for fetching the data does not saturate. In addition, to have high throughput, pages should be fetched with low latency. In LEADS, as crawling is performed by geographically distributed crawlers, pages will be fetched by crawlers that are located close to the servers holding them, and make good use of available bandwidth to ensure the throughput of crawling.

6.1.2.2 Dealing with malicious sites

There exist unexpected events that may prevent crawlers from efficient crawling. One possibility is crawler trap [HN99]. That is a web page or a set of URLs intentionally or even unintentionally used to make a crawler to generate infinite number of requests to fetch data from a web site. This would waste the resources of the crawler and thus reduces its efficiency for collecting high quality data. A poorly designed crawler may even crash under such circumstance. Another possibility is the delay attack [LLW+08], where some web sites purposely introduce HTTP and DNS delays in all requests of fetching pages that originate from the crawler's IP address. This would make the crawling latency unnecessarily high. Therefore, these factors need to be taken into account while designing an efficient and robust crawler. In LEADS, standard techniques like politeness to the sites and robot exclusion protocols may be used to avoid crawler trap while delay attack may be detected and avoided by abandon long latency crawling requests.

6.1.3 Requirements on cost

6.1.3.1 Computation cost

LEADS aims at spreading the collection, storage and processing of data across multiple micro-clouds geographically distributed to reduce the total cost of owning this platform. The computational cost related to data collections consists of the cost of maintaining the frontiers, fetching the pages, parsing the fetched pages and determining where to crawl the newly discovered pages. As we know, the

location of data centers can drastically affect the total cost of ownership through the differences in electricity price, labor cost, weather, etc. [AFG10]. The distributed crawlers of LEADS, located in different micro-clouds, should leverage its distributed nature to design crawling strategy that appropriately distributes the crawling tasks among crawlers. The cost for maintaining the frontier, fetching the pages and parsing the fetched pages highly depends on the pages to be crawled by a crawler. Therefore, making a wise decision on which crawler should crawl which page is key to maintain the overall computational cost of the crawling-based data collection low.

6.1.3.2 Communication cost

Collecting the data through crawling requires to fetching the pages linked by the URLs in the frontier. Therefore, load balancing among the distributed crawlers is very important to avoid some crawlers already saturate their bandwidth for fetching the pages, while other crawlers are idly running. Load balancing in this context refers to assigning amounts of workload to crawlers proportionally to the network resources they have. Moreover, as the Web is an inter-connected graph where pages are linked to other pages, it is unavoidable that links discovered by a crawler may appear to be more appropriate to be crawled by other crawlers. This would be particularly the case if data is placed in different micro-clouds according to specific applications that LEADS provides. In this case, URLs needs to be exchanged among crawlers, as shown in Figure 17, to ensure that they are always crawled by the most appropriate crawler.

6.1.4 Additional requirements

6.1.4.1 Politeness

While pursuing efficiency of crawling, the crawlers should be as polite as possible to Web servers from which they fetch the pages [Eei94]. This because requesting a Web server in high pace may incur a burden on that server, eventually degrading the service quality the corresponding Web site provides and causing it to consider the crawler as a hostile program. Hence, crawlers need to avoid overloading the web servers, while not sacrificing their own throughput. Typically, this can be achieved by limiting the number and duration of open network connections to the Web servers [BCS+04, CGP98, LLW+08]. The distributed crawlers of LEADS will follow these approaches to ensure politeness.

6.1.4.2 Robot exclusion protocol

Web sites often use a robots.txt file to explicitly inform web crawlers what are the pages they would like the crawlers to crawl and what are the pages they would not like the crawlers to crawl[6]. The crawler must maintain a cache of these files. When crawling a page, the robots.txt of the web site where the page is from must be checked. Respecting the robot exclusion protocol may lower the throughput of the crawler by performing such checks. However, this provides the crawlers an opportunity to avoid crawler trap. This is because sites with crawler traps usually have a robots.txt to tell the crawlers to not go to the trap. An impolite crawler is thus more likely to be affected by crawler traps than a polite crawler. Hence, The distributed crawlers of LEADS will respect the robot exclusion protocols for both politeness and efficient crawling.

6.2 Design of a web crawler

The crawling-based data collection system of the LEADS platform relies on multiple crawlers located in geographically distributed micro-clouds to perform crawling in a collaborative way. We describe in this section the high-level design of a single crawler instance, which is the basis of the entire system.

[6] A Standard for Robot Exclusion. http://www.robotstxt.org/orig.html.

Each crawler fetches the pages that are assigned to it according to the specific strategy deployed. Figure 18 shows the architecture of a crawler. Once a page is fetched, it is content is put into the Content Repository. The Parser extracts more URLs that are linked by this page and put the associated links to the Link Database. The Content and Links Repository uses storage layer envisioned by WP2 of LEADS. The URL filters then filter the newly obtained URLs by removing duplications, performing normalization etc. to only keep new URLs that are valid. The newly discovered URLs are then added to the URL Database, which is to track all the discovered URLs. The URL Prioritizor gets URLs from the URL Database and defines the crawling order as we discussed in Section 6.1.1.1 to ensure high quality pages are fetched first in the Frontier of the crawler. The DNS Resolver then identifies the IP address of the Web site hosting the page to fetch so that the Fetcher can get the page efficiently. Finally the robots.txt Checker verifies whether the discovered URLs should not be crawled as specified by the robot exclusion protocol as we discussed in Section 6.1.4.2, before further fetching the data linked by these URLs.

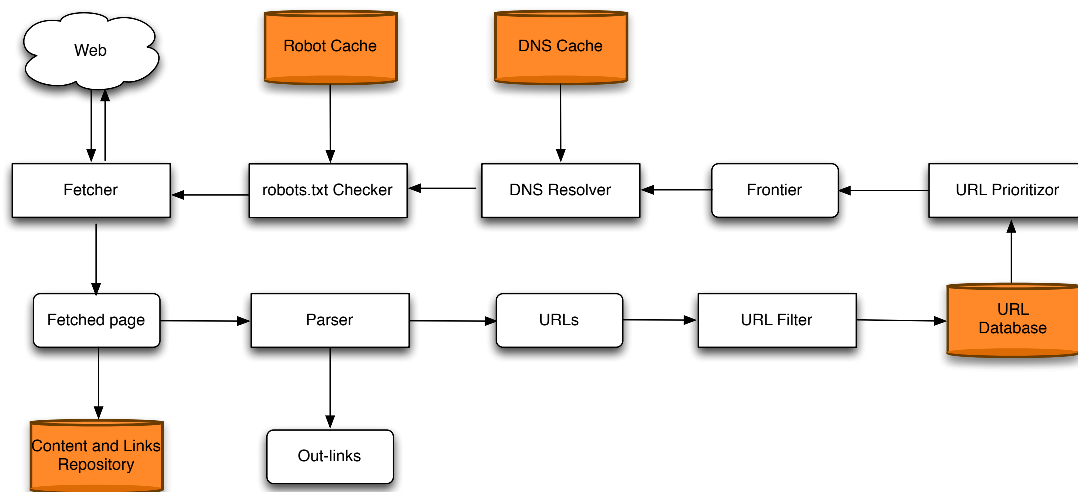


Figure 18: Architecture of a web crawler

6.3 Current implementation of the prototype crawling-based data collection

In this section, we present a prototype implementation of the crawler of LEADS. This prototype allows a set of distributed crawlers to collaboratively fetch data from the Web, which constitutes an crawling-based collection of web pages with the retrieved content. This prototype is publicly available[7]. It is built on top of Infinispan[8] and flaxcrawler[9]. Infinispan is a distributed key-value store maintained by the LEADS partner Red Hat, which implements the core of the LEADS storage layer. Flaxcrawler is an open-source lightweight and multi-threaded web crawler written in Java.

Below, we detail the capabilities of our prototype, its key components, as well as, the internal workflow when it executes the query specified in the Use Case 1 of LEADS [UCM12].

[7] Demo of LEADS crawler. http://github.com/otrack/Leads-crawler-demo.
[8] Infinispan. http://www.jboss.org/infinispan.
[9] Flaxcrawler. http://code.google.com/p/flaxcrawler/.

6.3.1 Capabilities

Our prototype keeps track of the visited web pages, so no page would be crawled twice. Specifically, before crawling a page linked by a URL, the crawler checks whether this URL exists in the index of the Link Database (Figure 18) or not. If it is the case, the URL is simply ignored. Otherwise, the crawler fetches its content. This ensures the efficiency of crawling by avoid redundant data fetching. The prototype normalizes the URLs of the crawled web pages, and it can be configured to make a pause between two requests to a single site with respect to the politeness policy. The key feature of our prototype is being both multi-threaded and distributed. In other words, the amount of crawlers that constitute the crawling-based collection can scale vertically (with more threads) and horizontally (with more servers). Our prototype supports downloading the pages through one or more proxies, and it can balance the load between proxies when the number of proxies is set to more than one. It is also highly customizable as the controllers for the download and the web page parsers can be defined by the user.

6.3.2 Components

The architecture of our crawler prototype follows the guidelines depicted in Figure 18. The key component is the content repository. This repository is implemented as a shared map on top of Infinispan. Each crawled page is stored in this repository, and is indexed accordingly to its URL.

After a page is crawled, the parser extracts the URLs from the page. For each URL found into the page, a filtering mechanism is applied to know if the URL is pushed to the link database. The parsing is configurable. For instance, the user can specify that a URL is not kept if it is dynamic, or it does not belong to a predefined set of domains.

The crawler implements a test to verify that a URL was not previously crawled. To this end, it checks the existence of the normalized URL in the content repository. Notice here that the shared map implementing the content repository is not strongly consistent. This costs that a small percentage of URLs can be re-crawled erroneously, but this significantly improves the whole performance of the system.

Once the filtering mechanism acknowledges that a URL can be crawled, it is stored into the link database. This database is implemented via an atomic queue on top of Infinispan. By executing a *pop()* operation on the queue, a crawler retrieves the next URL it will have to crawl. To add a URL to the database, a crawler executes a *push()* operation. In the initial state, the queue is populated with the seed, which is the set of URLs that define the start of the crawl.

6.3.3 Internal workflow for Use Case 1

In the Use Case 1 of LEADS, a user provides a list of keywords to compute a query on top of the publicly available data in LEADS. The result of the query is a list of web pages that match the provided keywords, and that are sorted according to both their PageRank and to a sentiment analysis score of their content. A video of demonstration of the query is available online[7].

To implement the query, the LEADS infrastructure splits the query in two parts:

- A permanent part, which runs continuously and constructs a crawling-based data collection, and

- An instantaneous part, which returns the state of the result at the time and is executed on top of this data collection.

The permanent part is executed by our prototype of distributed crawlers. The set of pages resulting from this continuous crawl of the Web is parsed by a set of listeners that check if the content matches the key words. In case, the content matches, the listener retrieves a sentiment analysis score of the page from a remote service and stores the resulting page in a shared mapping together with the PageRank of the web sites hosting the page. When the instantaneous part of the query is executed, the content of the map is sorted appropriately and then returned to the user.

7. Conclusion

This deliverable focused on the user-aided data collection mechanism of LEADS, and detailed the design and implementation of the user-aided publishing interface. It also presented experimental evaluation on large-scale datasets, showing that the user-aided data collection could be a good complement to the crawling-based data collection to enlarge the data collection of LEADS. Finally, this deliverable gave a brief overview of the prototype of the distributed crawlers.

8. References

[AFG10] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. Communications ACM, 53(4), 2010.

[ATT+09] E. Adar, J. Teevan, S. T. Dumais, and J. L. Elsas. The web changes everything: understanding the dynamics of web content. In Proc. 2nd ACM Int'l Conf. on Web Search and Data Mining, pages 282–291, 2009.

[BCS+04] P. Boldi, B. Codenotti, M. Santini and S. Vigna. UbiCrawler: a scalable fully distributed Web crawler. Software, Practice & Experience vol. 34 (8) pp. 711-726, 2004.

[Ber01] M. K. Bergman. White paper: the deep Web: surfacing hidden value. The Journal of Electronic Publishing, 7(1):online, 2001.

[BF07] L. Barbosa and J. Freire. An adaptive crawler for locating hidden-web entry points. In Proc. 16th Int'l Conf. on World Wide Web, pages 441–450, 2007.

[BC05] Ricardo Baeza-Yates and Carlos Castillo. Crawling the Infinite Web. Journal of Web Engineering, 2005.

[BOM12] M. Boanjak, E. Oliveira, J. Martins, E. M. Rodrigues and L. Sarmento. TwitterEcho: a distributed focused crawler to support open research with twitter data. In Proc. of the 21st Int'l Conf. Companion on World Wide Web, pages 1233-1240, 2012.

[CDF+11] S. Catanes, P. De Meo, E. Ferrara, G. Fiumara and Alessandro Provetti. Crawling Facebook for social network analysis purposes. In Proc. of the Int'l Conf. on Web Intelligence, Mining and Semantics, page 52, 2011.

[CG00] J. Cho and H. Garcia-Molina. The evolution of the Web and implications for an incremental crawler. In Proc. 26th Int'l Conf. on Very Large Data Bases, pages 200–209, 2000.

[CG02] J. Cho and H. Garcia-Molina. Parallel crawlers. In Proc. 11th Int'l Conf. on World Wide Web, pages 124–135, 2002.

[CG03] J. Cho and H. Garcia-Molina. Effective page refresh policies for Web crawlers. ACM Trans. Database Syst., 28(4):390–426, 2003.

[CGP98] J. Cho, H. Garcia-Molina, and L. Page. Efficient crawling through URL ordering. Comput. Netw. ISDN Syst., 30(1-7):161–172, 1998.

[CPJ+08] B. B. Cambazoglu, V. Plachouras, F. Junqueira, and L. Telloli. On the feasibility of geographically distributed Web crawling. In Proc. 3rd Int'l Conf. on Scalable Information Systems, pages 1–10, 2008.

[D2.2] Deliverable 2.2 of LEADS: Initial prototype of the key-value store.

[DGK+07] A. Dasgupta, A. Ghosh, R. Kumar, C. Olston, S. Pandey, and A. Tomkins. The discoverability of the Web. In Proc. 16th Int'l Conf. on World Wide Web, pages 421–430, 2007.

[Eei94] D. Eichmann. Ethical web agents. In Proc. Of 2nd Int'l Conf. on World Wide Web, pages 3–13, 1994.

[EMP+05] J. Exposto, J. Macedo, A. Pina, A. Alves, and J. Rufino. Geographical partition for distributed Web crawling. In Proc. 2005 Workshop on Geographic Information Retrieval, pages 55–60, 2005.

[EMP+08] J. Exposto, J. Macedo, A. Pina, A. Alves, and J. Rufino. Efficient partitioning strategies for distributed Web crawling. In Information Networking. Towards Ubiquitous Networking and Services, volume 5200 of Lect. Notes Comput. Sc., pages 544–553, 2008.

[EMT01] J. Edwards, K. McCurley, and J. Tomlin. An adaptive model for optimizing performance of an incremental web crawler. In Proc. 10th Int'l Conf. on World Wide Web, pages 106–113, 2001.

[EMT04] N. Eiron, K. S. McCurley, and J. A. Tomlin. Ranking the web frontier. In Proc. 13th Int'l Conf. on World Wide Web, pages 309–318, 2004.

[FCV09] D. Fetterly, N. Craswell, and V. Vinay. The impact of crawl policy on web search effectiveness. In Proc. 32nd Int'l ACM SIGIR Conf. on Research and Development in Information Retrieval, pages 580–587, 2009.

[FMN+04] D. Fetterly, M. Manasse, M. Najork, and J. L. Wiener. A large-scale study of the evolution of web pages. Softw. Pract. Exper., 34(2):213–237, 2004.

[JKK+08] J. Madhavan, D. Ko, L. Kot, V. Ganapathy, A. Rasmussen, and A. Halevy. Google's deep web crawl. Proc. VLDB Endowment, 1(2):1241–1252, 2008.

[KGM07] G. Koutrika and H. Garcia-Molina. Fighting spam on social Web sites: a survey of approaches and future challenges. IEEE Internet Computing: 11(6):36—45, 2007.

[HN99] A. Heydon and M. Najork. Mercator: a scalable, extensible Web crawler. Proc. 8th Int'l Conf. on World Wide Web, 2(4):219–229, 1999.

[LG00] S. Lawrence and C. L. Giles. Accessibility of information on the Web. Intelligence. 11(1):32–39, 2000.

[LLW+08] H.-T. Lee, D. Leonard, X. Wang, and D. Loguinov. IRLbot: scaling to 6 billion pages and beyond. In Proc. 17th Int'l Conf. on World Wide Web, pages 427–436, 2008.

[LNJ+10] E.-P. Lim, V.-A. Nguyen, N. Jindal, B. Liu and H. W. Lauw. Detecting product review spammers using rating behaviors. In Proc. of the 19th Int'l Conf. on Information and knowledge management, pages 939—948, 2010.

[NCO04] A. Ntoulas, J. Cho, and C. Olston. What's new on the web?: the evolution of the web from a search engine perspective. In Proc. 13th Int'l Conf. on World Wide Web, pages 1–12, 2004.

[NW01] M. Najork and J. L. Wiener. Breadth-first crawling yields high-quality pages. In Proc. 10th Int'l Conf. on World Wide Web, pages 114–118, 2001.

[NZC05] A. Ntoulas, P. Zerfos, and J. Cho. Downloading textual hidden web content through keyword queries. In Proc. 5th ACM/IEEE-CS Joint Conf. on Digital Libraries, pages 100–109, 2005.

[PO05] S. Pandey and C. Olston. User-centric web crawling. In Proc. 14th Int'l Conf. on World Wide Web, pages 401–411, 2005.

[RGM01] S. Raghavan and H. Garcia-Molina. Crawling the hidden web. In Proc. 27th Int'l Conf. on Very Large Data Bases, pages 129–138, 2001.

[SO05] S. Pandey and C. Olston. User-centric web crawling. In Proc. 14th Int'l Conf. on World Wide Web, pages 401–411, 2005.

[SS02] V. Shkapenyuk and T. Suel. Design and implementation of a high-performance distributed Web crawler. In Proc. 18th Int'l Conf. on Data Engineering, page 357, 2002.

[UCM12] LEADS M12 accompanying document: use cases specification.

[ZD02] D. Zeinalipour-Yazti and M. D. Dikaiakos. Design and implementation of a distributed crawler and filtering processor. In Proc. 5th Int'l Workshop on Next Generation Information Technologies and Systems, 2002.