System Architecture Design and Implementation

1 The Toolkit

In this document we present the design and implementation of our toolkit, whose components and workflow are summarized in Figure 1. It consists of a core application for massive Web crawling, indexing and text mining, and of two external independent modules for customized/focused crawling and structure mining respectively.

1.1 The Core Application

At the heart of the toolkit is a core application in charge of automatically exploring Tor websites and collecting their contents, while indexing and clustering gathered data. Based on an analysis of other applications oriented to the collection and analysis of data, we designed the core of our toolkit around four main components, all written in Java: a coordinator, a crawler, an extractor and an analyser.

In the following, we describe the toolkit's core application more in detail: we start with an overview of its components and workflow, that prompts us to report on implementation and operational aspects; we then summarize configuration options, pointing out the most important ones; finally, we focus on the crawler, discussing its design and functionalities.

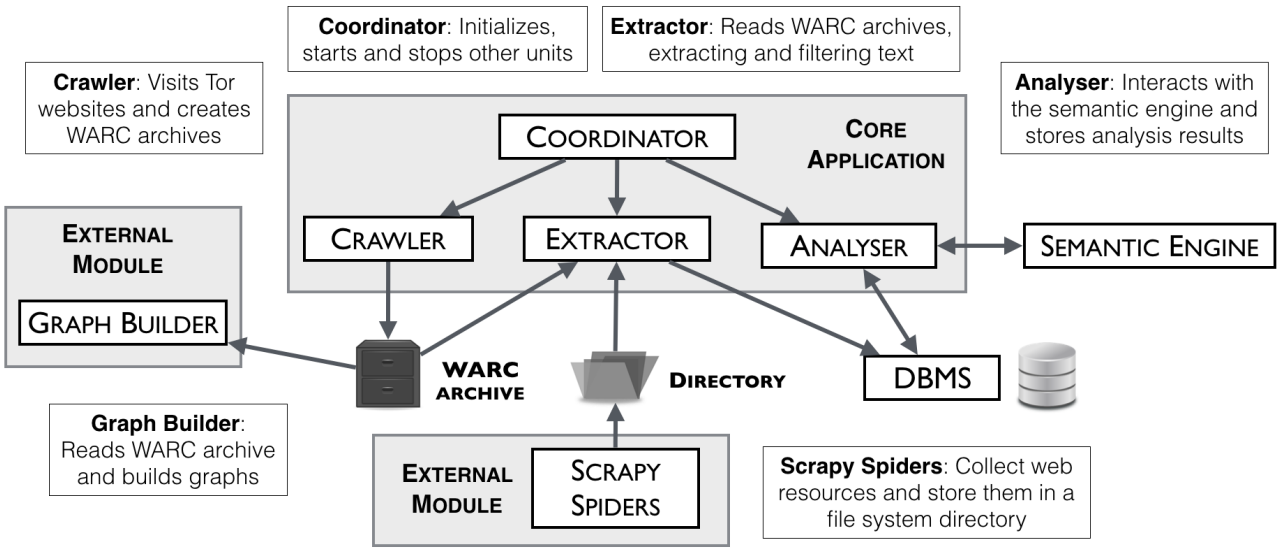

Figure 1: Components and workflow.

1.1.1 Components and Workflow

The main process of the core application, that we developed from scratch, is the coordinator, responsible of organizing the operations of other units. When launching the tool, the coordinator is activated and it starts reading the configuration file and setting up the application. It checks the existence of the database and creates one if it does not exist. After initialization procedures, the coordinator activates the crawler, the process which concretely performs the task of visiting Tor websites.

The crawler carries out a breadth-first search of Tor websites and stores all retrieved resources in a WARC archive. A list of known hidden services, the *root set*, is provided to the crawler to determine the data collection starting points. When the archive reaches a threshold size, the coordinator stops the crawler. The threshold size of the archive is a configuration parameter, which can be modified in the configuration file. Once the crawler stops, the coordinator moves the WARC archive and creates a new empty file, which will be used by the crawler at its next activation, then the extractor starts.

The extractor is a multi-thread process that concurrently reads and extracts text from resources contained in the WARC archive (and/or in a file system directory, see Section 1.1.2). The extractor reads the WARC archive, extracts texts from collected data and stores them in a database. To extract text from digital documents, the extractor uses the Apache Tika API [1]. Apache Tika is a project of the Apache Software Foundation, providing a java toolkit able to detect and extract metadata and text from over a thousand different file types. The extractor only stores texts from web pages that replied successfully during the crawling (HTTP status code "200 OK"), and it filters extracted texts according to specific configuration options (*e.g.* on a language basis). All data are stored in a document-oriented database, in which each web resource is stored as a document. For each web resource the extractor stores: (i) the HTTP response header, (ii) the WARC record header, (iii) the metadata provided by Tika, (iv) the extracted text, and (iv) the language of the document's text. Specifically, we use ArangoDB [2], a multi-model, open-source, NoSQL database with flexible data models for documents, graphs, and key-values. It supports ACID transactions if required and provides a SQL-like query language or JavaScript extensions. Once the extractor terminates its operations, the coordinator activates the analyser.

The analyser is a multi-thread process that concurrently reads documents from the database and sends their texts to the semantic engine, for analysis. For each document the analyser prepares a RESTful request, containing its text and language, and sends the request to the engine. The engine sends back analyses results, which are again stored by the analyser in the initial database, together with other document's information. Once all documents have been analysed, the coordinator stops the analyser.

The analyser relies on Cogito, a multi-language semantic engine developed by Expert System, which can understand the meaning in context within unstructured text. Cogito is able to find hidden relationships, trends and events, transforming unstructured data into structured information. Through several analyses, it identifies three different types of entities (people, places and companies/organizations), categorizes documents on the basis of several taxonomies and it is able to extract entity co-occurrences.

[1] https://tika.apache.org

[2] https://www.arangodb.com

1.1.2 Configuration

The application behaviour can be set up by modifying a Java properties file, named "config.properties" . First of all, through the configuration file it is possible to determine what operations the application will perform, by selecting any subset of the three building blocks, *i.e.* crawling, extraction, and analysis. If the extractor is enabled, either or both of two supported modalities can be selected namely extraction from WARC and extraction from file. In the first modality the extractor uses a WARC file, in the latter it uses a file system directory to retrieve resources to elaborate.

A number of options can be specified through the configuration file, including: (i) the number of threads used by the extractor and the analyser agents; (ii) the threshold size of WARC archives generated by the crawler – when that size is reached, the archive is passed to the extractor and a new archive is created by the crawler; (iii) the directory used by the crawler to store data; (iv) the name of the database to be created or used to store documents; (v) the name of the database's collections used by the extractor and the analyser agents; (vi) the directory used by the extractor to retrieve file – in case the extraction from file modality has been activated. Moreover, it is possible to activate the file system data storage option, *i.e.* data produced by crawler, extractor and analyser can be stored locally in the file system, keeping in mind that a copy of the same data is stored in the database by default. Finally, the configuration file contains the address and other parameters used to contact the RESTful web service exposed by the semantic engine.

1.1.3 The Crawler

We evaluated different alternatives for the development of the crawler. In particular, we implemented a crawler prototype from scratch and evaluated it against three main existing candidates: Apache Nutch [3] [2], Heritrix [4] [3] and BUbiNG [1]. Considering several criteria, such as performance, configurability, extensibility and supportability, we found BUbiNG to be the most appropriate choice as the base for our crawler component. BUbiNG is a high-performance, scalable, distributed, open-source crawler, written in Java, and developed by the Laboratory for Web Algorithmics (LAW) at the Computer Science Department of the University of Milan.

Significant efforts were needed for the integration of BUbiNG within our toolkit, so as to turn it into an application's component under the control of the coordinator. Moreover, we needed to enable BUbiNG operating in Tor instead of the surface Web. Whereas, by default, BUbiNG presents a set of threads that perform DNS requests, in our application we avoid these requests and send them directly to a HTTP proxy. We chose to use privoxy [5], after testing other alternatives. In particular, we decided not to use polipo [6], that is often used in combination with Tor, because it is no longer maintained and currently seems not able to manage correctly the format of some HTTP responses. However, any HTTP proxy configured to use Tor can be used. Through the crawler configuration file, that is a Java properties file named 'crawler.properties', it is possible to specify which HTTP proxy the crawler must use. Several other parameters can be set through that configuration file, including: (i) the number

[3] http://nutch.apache.org

[4] https://webarchive.jira.com/wiki/display/Heritrix

[5] https://www.privoxy.org

[6] https://www.irif.fr/ jch/software/polipo/

of threads the crawler will use for its operations (parsing, dns resolution, fetching); (ii) which resources are collected from websites (*e.g.* html pages, media file, digital documents); (iii) network timeouts used when contacting websites; (iv) where collected date will be stored; (v) how and whether to manage cookies; (vi) the delays between requests to the same website. For a complete list of configuration parameters we refer the reader to BUbiNG's documentation [7]. With our toolkit we provide a standard crawler configuration, tested for Tor network, thus editing that file is not needed for standard usage.

For what concerns the crawling process, in BUbiNG a pool of software agents are responsible for both exploring the Web and collecting (and partially elaborating) the data. Such agents work in parallel, each handling in turn several threads, and by default implementing a *breadth-first-search*. Due to Tor's high volatility, the results of the crawling process are theoretically susceptible to fluctuations based on the order in which links are followed. Yet, we did not have any reason to prefer one order over another, and we therefore did not modify this setting. Summing up, in our experiments BUbiNG was used as follows:

- A predetermined set of hidden services, the *root set*, is inserted in an *url list*.
- The first *fetching thread* available extracts the first onion url from the list and exports the content of the corresponding hidden service in main memory.
- The first *parsing thread* available analyses that content aiming at extracting new onion urls to visit.
- The new onion urls found are passed to a *sieve* able to verify whether those urls were already visited before.
- If so, the urls are discarded, otherwise they are added in the tail of the url list.
- *Fetching threads* attempt to contact each url for a maximum of three times, after that the url is considered not available.

1.2 External Modules

Alongside of the core application, our toolkit comprehends two external modules which can be launched independently: a set of Scrapy Spiders, written in Python, and a Graph Builder, written in C. The spiders enhance the crawling process by permitting focused crawling, supporting semi-automated procedures, and offering anti-detection settings. The Graph Builder supports the reconstruction of the graph associated to the crawling process, enabling topological studies of the explored portion of the Web.

1.2.1 Scrapy Spiders

Besides BUbiNG, we developed and integrated in our toolkit a set of customized spiders which can be managed as modules and used to boost the crawling process. Specifically, our spiders were written relying on Scrapy [8], a Python framework for crawling websites and extracting data.

[7] http://law.di.unimi.it/software/bubing-docs/overview-summary.html
[8] https://scrapy.org

Using ad-hoc spiders besides the main crawler allows for focused crawling which supports a semi-automated (*i.e.* human assisted) procedure needed to gain access to hidden contents requiring a login procedure or a captcha solver. Moreover, through configuration settings, we are able to choose a breadth-first or depth-first search strategy for the crawling procedure. A further customization required for our spiders consists in avoiding crawling detection, which may be implemented by our targets. To that purpose, we included configuration settings concerning the robots exclusion protocol, network timeouts and (again) delays between requests to the same website. Furthermore, we programmed our spiders to visit only specific sections of targeted websites and to carry out only legal and not suspicious actions. Finally, the data collected by Scrapy Spiders are stored as files in a directory and are integrated in the core application via the extractor module. Indeed, through configuration settings, the extractor unit can be instructed to retrieve files from system directories. The Scrapy Spiders module can be used as a template to create other spiders with the same framework, or as an example of how to integrate data collected by other crawlers.

1.2.2 Graph Builder

The Graph Builder is written in C and supports the reconstruction of the graphs associated to the crawling process, using the WARC archive created by BUbiNG. To parse HTML file and extract links we use the mythml library [9]. The Graph Builder module is a multi-thread application, it takes as input the name of the WARC file to read and the number of threads to use to build the graphs. For each WARC file three directed graphs are created by the module, a page graph, a host graph and a service graph, which are represented as list of edges. In the page graph an edge exists between page A and page B if there is a least a link from page A to page B. In the host graph an edge exists between host A and host B if there is a least a page of host A that links to a page of host B. The service graph is the higher level of grouping, in which each node represents an hidden service, which is identified by a 16 character string (base32 encoded). In this graph an edge exists between service A and service B if there is a least a page of service A that links to a page of service B. Each graph is represented by two files that are written as output by the module: a ".index" and a ".edges" file. The ".index" file contains the id of each graph's node with the corresponding url, while the ".edges" file contains the list of graph's edges represented as id pairs.

References

[1] P. Boldi, A. Marino, M. Santini, and S. Vigna. Bubing: Massive crawling for the masses. In *Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web Companion*, pages 227–228, 2014.

[2] R. Khare, D. Cutting, K. Sitaker, and A. Rifkin. Nutch: A flexible and scalable open-source web search engine. *Oregon State University*, 1:32–32, 2004.

[3] G. Mohr, M. Stack, I. Ranitovic, D. Avery, and M. Kimpton. An introduction to heritrix an open source archival quality web crawler. In *In IWAW04, 4th International Web Archiving Workshop*. Citeseer, 2004.

[9] https://github.com/lexborisov/myhtml