Client-Side Deep Web Data Extraction *

Manuel Álvarez, Alberto Pan+, Juan Raposo, Angel Viña
Department of Information and Communications Technologies
University of A Coruña – 15071 A Coruña - Spain
{mad, apan, jrs, avc}@udc.es

Abstract

The problem of data extraction from the Deep Web can be divided into two tasks: crawling the client-side and the server-side deep web. The objective of this paper is to define an architecture and a set of related techniques to access the information placed in the client-side deep web. This involves dealing with aspects such as JavaScript technology, non-standard session maintenance mechanisms, client redirections, pop-up menus, etc. Our work uses current browser APIs as building blocks and leverages them to implement novel crawling models and algorithms.

1. Introduction

The "Hidden Web" or "Deep Web" [4] is usually defined as the part of WWW documents that is dynamically generated. The problem of crawling the "hidden web" can be divided in two tasks: crawling the client-side and the server-side hidden web. Client-side techniques are concerned about accessing content dynamically generated in the client web browser, while server-side techniques are focused in accessing to the valuable content hidden behind web search forms[1][7]. This paper proposes novel techniques and algorithms for dealing with the first of these problems.

1.1. The case for client-side hidden web

Today's complex web pages intensively use scripting languages (mainly JavaScript), session maintenance mechanisms, complex redirections, etc.

Developers use these client technologies to add interactivity to web pages as well as for improving site navigation. Most of the tools used for visually building web sites do it too.

1.2. The problem with conventional crawlers

There exist some problems that make difficult for traditional web crawling engines to obtain data from client-side hidden web pages. The most important problems are enumerated below (see [10] for more detail):

- Successfully dealing with scripting languages requires that HTTP clients implement all the mechanisms that make possible to a browser to render a page and to generate new navigations.
- Many websites use session maintenance mechanisms based on client resources like cookies or scripting code to add session parameters to the URLs before sending them to the server. This provokes problems when distributing the crawling process and for later access to the documents.
- Many websites use complex redirections that are not managed by conventional crawlers – e.g. JavaScript redirections -.
- Other types of client technology that conventional crawlers are not able to deal with are applets and flash code.

The aforementioned problems are accentuated by issues such as frames, dynamic HTML or HTTPS. We can say that it is very difficult to consider all the factors which make a Website visible and navigable.

1.3. Our approach

Due to all the reasons mentioned above, many designers of web sites avoid those practices in order to make sure their sites are on good terms with the crawlers. Nevertheless, this forces them to either increment the complexity of their systems by moving functionality to the server, or reducing interactivity

* This research was partially supported by the Spanish Ministry of Science and Technology under project TIC2001-0547.
+ Alberto Pan's work was partially supported by the "Ramón y Cajal" programme of the Spanish Ministry of Science and Technology.

with the user. Neither of these situations is desirable: web site designers should think in terms of "improving interactivity and friendliness of sites", not about "how the crawlers work".

This paper presents architecture and a set of related techniques to solve the problems involved in crawling the client-side hidden web. The main features of our approach are the following:

- To solve the problem of session maintenance, our system uses the concept of *route* to a document, which can be seen as a generalization of the concept of URL (section 3.1).
- To deal with executing scripting code, managing redirections, etc., our crawling processes are not based on http clients. Instead, they are based on automated "mini web browsers", built using standard browser APIs (section 3.2).
- To deal with pop-up menus and other dynamic elements that can generate new anchors in the actual page, it is necessary to implement special algorithms to manage the process of generating new "routes to crawl" from a page (section 3.4).
- The system also includes some functionality to access pages hidden behind forms. More precisely, it is able to deal with authentication forms and with *value-limited forms*. We term as *value-limited* forms those ones exclusively composed of fields whose possible values are restricted to a certain finite list.

Section 3 describes the architecture and basic functioning of the system. See the extended version of this paper [10] for more information.

2. Introduction to NSEQL

NSEQL (Navigation SEQuence Language [2]) is a language to declaratively define sequences of events on the interface provided by a web browser. NSEQL allows to easily expressing "macros" representing a sequence of user events over a web browser.

NSEQL works "at browser layer" instead of "at HTTP layer". This lets us forget about problems such as successfully executing JavaScript or dealing with client redirections and session identifiers.

3. The crawling engine

As well as in conventional crawling engines, the basic idea consists in maintaining a shared list of routes (pointers to documents), which will be accessed by a certain number of concurrent crawling processes, which may be distributed into several machines. The list is initialized and then, each crawling process picks a route from the list, downloads its associated document and analyzes it for obtaining new routes from its anchors, which are then added to the master list. The process ends when there are no routes left or when a specified depth level is reached.

3.1. Dealing with sessions: Routes

In conventional crawlers, routes are just URLs. In our system, a route is composed of three elements: a) A URL pointing to a document. In the routes from the initial list, this element may also be a NSEQL program; b) A session object containing all the required information (cookies, etc.) for restoring the execution environment that the crawling process had in the moment of adding the route to the master list; c) A NSEQL program representing the navigation sequence followed to reach the document.

The second and third elements are automatically computed by the system for each route. The second element allows a crawling process to access a URL added by other crawling process. The third element is used to access the document pointed by the route when the session originally used to crawl the document has expired or to allow later access to them.

3.2. Mini-browsers as crawling processes

Conventional engines implement crawling processes by using http clients. Instead, the crawling processes in our system are based on automated "mini web browsers", built using standard browser APIs (our current implementation uses the MSIE – Microsoft Internet Explorer [9] -WebBrowser Control). These "mini-browsers" understand NSEQL. This allows our system to:

- Access the content dynamically generated through scripting languages.
- Evaluate the scripting code associated with anchors and forms, for obtaining their real URLs.
- Deal with client redirections: after executing a navigation step, the mini-browser waits until all the actual page navigation events have finished.
- Provide an execution environment for technologies such as Java applets and Flash code. Although the mini-browsers cannot access the content of these "compiled" components, they can deal with the common situation where these components redirect the browser to a conventional page after showing some graphical animation.

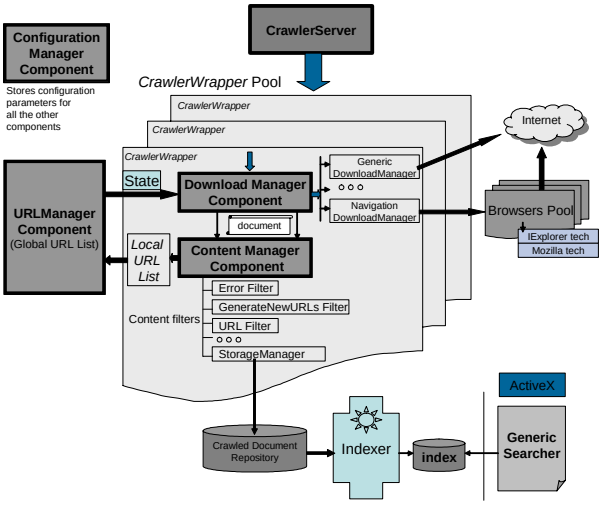

Figure 1. Crawler Architecture

3.3. System architecture / Basic functioning

The architecture of the system is shown in Figure 1.

When the crawler engine starts, it reads its configuration parameters from the Configuration Manager module (it includes a list of initial URLs and/or NSEQL navigation sequences, the desired depth for each initial route, download handlers for different kinds of documents, content filters, DNS domains to include and to exclude, etc.).

The following step consists in starting the URL Manager Component with the list of initial sites to crawl, as well as the pool of crawling processes (locally or remotely to the server).

The URL Manager is responsible of maintaining the master list of *routes* to be accessed, and all the crawlers share it. As the crawling proceeds, the crawling processes add new routes to the list by analyzing the anchors and value-limited forms found in the crawled pages. Once the crawling processes have been started, each one picks a route from the URL Manager. Then the crawling process loads the session object associated to the route and downloads the associated document (it uses the Download Manager Component to choose the right handler for the document). If the session has expired, the crawling process will use the NSEQL program instead.

The content from each downloaded document is then analyzed using the Content Manager Component. This component specifies a chain of filters to decide if the document can be considered relevant and, therefore, if it should be stored and/or indexed. For instance, the system includes filters which allow checking if the document verifies a keyword-based boolean query with a minimum relevance, in order to decide whether to store/index it or not. Another chain of filters is used for post-processing the document – for instance, the HTML content extractor or the document summary generator-.

Finally, the system tries to obtain new routes from analyzed documents and adds them to the master list. In a context where scripting languages can dynamically generate and remove anchors and forms, this involves some complexities (see section 3.4).

The system also includes a chain of filters to decide whether the new routes must be added to the master list or not.

The architecture has components for indexing and searching the crawled contents, using state of the art algorithms. When the user makes a search against the index and the list of answers contains some results which cannot be accessed using its URL due to session issues, the anchors associated to those results in the list will invoke the ActiveX for automatic navigation Component. It receives as a parameter a NSEQL program, downloads itself into the user browser and makes it execute the given navigation sequence.

3.4. Algorithm for generating new routes

This section describes the algorithm used to generate new routes to be crawled given a HTML page. This algorithm deals with the difficulties associated to anchors and forms controlled by scripting languages.

In general, to get the new routes to be crawled from a given HTML document, it is necessary to analyze the page looking for anchors and *value-limited* forms. A new route will be added for each anchor and for each combination of all the possible values of the fields from each *value-limited* form. The anchors and forms, which are not controlled by scripting code, can be dealt with as in conventional crawlers. Nevertheless, if the HTML page contains client-side scripting technology, the situation is more complicated. The main idea of the algorithm consists on generating click events over the anchors controlled by scripting languages in order to obtain valid URLs (NOTE: we will focus our discussion on the case of anchors. The treatment of value-limited forms would be analogous), but there are several additional difficulties:

– Some anchors may appear or disappear from the page depending on the scripting code executed.
– The script code associated to anchors must be evaluated in order to obtain valid URLs.
– One anchor can generate several navigations.

– In pages with several frames, it is possible for an anchor to generate new URLs in some frames and navigations in others.

Now we proceed to describe the algorithm. The crawler process can be in two states: in the *navigation* state the browser functions in the usual way; in turn, in the *simulation* state the browser only captures the navigation events generated by the click or submit events, but it does not download the resource.

1. Let P be an HTML page that has been downloaded by the browser (navigation state).
2. The browser executes the scripting sections which are not associated to conditional events.
3. Let A_p be all the anchors of the page with the scripting code already interpreted.
4. For each a_i Є A_p, remove a_i
   a) If the *href* attribute from a_i does not contain associated scripting code and it has not got an *onclick* attribute (or other attributes such as *onmouseover*, if defined), the anchor a_i is added to the master list of URLs.
   b) Otherwise, the browser generates a *click* event on the anchor (other event, if defined):
      a. There exist some anchors that, when clicked, can generate undesirable actions (e.g. a call to the "javascript:close" method closes the browser). The approach followed to avoid this is to capture and ignore these events.
      b. The crawler captures all the new navigation events that happen after the click. Each navigation event produces an URL. Let A_n be the set of all the new URLs.
      c. $A_p = A_n \cup A_p$.
      d. Once the execution of the events associated to a click over an anchor has finished, the crawler analyzes the same page again looking for new anchors that could have been generated by the click event (e.g., new options corresponding to pop-up menus), A_{np}. New anchors are added to A_p, $A_p = A_{np} \cup A_p$.
5. The browser changes to navigation state, and the crawler is ready to process a new URL.

If the processed page has several frames, then the system will process each frame in the same way.

Note that the system processes the anchors in a page following a bottom-up approach, so new anchors are added on the list before the existing ones. In this way, new anchors will be processed before some other click can remove them from the page. Also note that the added anchors will have to agree with the filters for adding URLs mentioned in section 3.3

4. Related work and conclusions

A well-known approach for discovering and indexing relevant information is to "crawl" a given information space looking for information verifying certain requirements. Nevertheless, today's web "crawlers" or "spiders" [3] do not deal with the hidden web. During the last few years, there have been some pioneer research efforts dealing with the complexities of accessing the hidden web [1][7] using a variety of approaches, but only concerned with server-side hidden web. Some crawling systems [8] have included JavaScript interpreters [5][6] in the HTTP clients they use in order to provide some support for dealing with JavaScript. Nevertheless, our system offer several advantages over them:

– It is able to correctly execute any scripting code in the same manner as a conventional web browser.
– It is able to deal with session maintenance mechanisms for both crawling and later access to documents (using the ActiveX component).
– It is able to deal with anchors and forms dynamically generated in response to user events.
– It is able to deal with redirections (including those generated by Java applets and Flash programs).

Finally, we want to remark that the system presented in this paper has already been successfully used in several real-world applications in fields such as corporate search and technology watch.

5. References

[1] S. Raghavan and H. García-Molina. "Crawling the Hidden Web". VLDB,2001.

[2] A. Pan, J. Raposo, M. Álvarez, J. Hidalgo and Angel Viña. "Semi-Automatic Wrapper Generation for Commercial Web Sources". EISIC,2002.

[3] S. Brin and L. Page. "The Anatomy of a Large-Scale Hypertextual Search Engine". WWW, 1998.

[4] M.K. Bergman. "The Deep Web. Surfacing Hidden Value". http://www.brightplanet.com/technology/deepweb.asp

[5] Mozilla Rhino - JavaScript Engine (Java). http://www.mozilla.org/rhino/.

[6] Mozilla SpiderMonkey – JavaScript engine (C) http://www.mozilla.org/js/spidermonkey/.

[7] P.G. Ipeirotis and L. Gravano. "Distributed Search over the Hidden Web: Hierarchical Database Sampling and Selection". VLDB. 2002.

[8] WebCopier – Feel the Internet in your Hands. http://www.maximumsoft.com/.

[9] Microsoft Internet Explorer WebBrowser Control, http://www.microsoft.com/windows/ie

[10] Client-Side Deep Web Data Extraction extended paper (http://www.tic.udc.es/~mad/publications/csdeepweb_extended.pdf)