4 Copying Websites

Xavier Roche

xavier@httrack.com

4.1 Introduction – The Art of Copying Websites

The fundamental difference between copying file structures or ftp sites, and websites, lies on the very deep nature of the World Wide Web. No "directory listing" in the HTTP protocol, nor bulk transfer of website zones: it is a design choice for the Web – a collection of heterogeneous resources, not necessarily related to each other. A collection of pages generated from a database content, for example, is an unstable realm of information – shall the database be flushed, it then disappears. The Web is fundamentally a moving form: ftp directories often change, but you can easily synchronize them to obtain a up-to-date state wherever you are. It is only a matter of data stored on a file repository. But a Web page is potentially unique: it can be a clock counter, a real-time information delivered on demand, a user-specific or session-specific view of a more general data collection. It can be anything you want: its internal logics are hidden to the navigator and its user. Simply speaking, an ftp server is a collection of files, more like a remotely accessible, public hard disk. A Web server is more a collection of logical resources delivering content to clients. These logical resources can be programs connected to databases, to other systems, with complex interactions with the user's preferences and needs and the server environment (database, external sources, current state, etc.). The remote client never sees this logic: only the resulting content is accessible.

Hence, there are three ways to copy entire websites (see Chap. 1 'Web Archiving Issues and Methods'): the first, server-side archiving is the hard one. It consist in contacting each Web masters and convince them to organize a copy of their internal information system files, database schemas and system specifications, and then set-up the same architecture – hardware, software, environment (such as external data sources that could be used). This solution, generally hard to deploy even for the Web masters themselves, cannot be seriously considered for a wide-range solution. The

second choice is to have this done close to the server and record all transactions (transactions archiving). The last one is to automatically collect the delivered information directly from websites, as a regular browser would do (client-side archiving).

It is a makeshift as mirrors will never be perfect: as taking a photograph of a moving scene, you will not be able to recreate its movement. You will not be anymore able to get the real-time feeling when browsing online temperatures reports or stock exchange movements. But it is an acceptable compromise in term of feasibility and quality in most cases – a static photography of a website, a photography that we would preserve in a photoalbum. A photography that could be viewed again and again without even bothering about the existence of the live model anymore.

Copying websites using this technique is something very intuitive: the method is the exactly the same as if you were copying a website by yourself, using a regular browser. You would start from the first page, save it, save the associated images, and then click on each links to view them, save the corresponding pages on disk, and carry on until you saved all the pages you wanted to copy. After that, make some changes inside the HTML pages so that they can be viewable locally by your browser, checking all relevant tags. But copying more that one or two pages manually is a bit tiresome, and an automated tool can be a relieving solution. The automated link enumerator is generally called the "parser," and the automated remote data downloader the "crawler." These two main components have additional roles: the parser is also responsible for ensuring that links will still work in a local copy, by changing the URL syntax to a compatible, a "fully relative" one; and the crawler is also responsible for handling caching and updates.

There are many reasons why you want to copy a website. At the national school of engineering of Caen, we wanted to archive small and medium websites, not for classical archiving, but to gather technical sites run by individuals which were moving very quickly. We also wanted to collect sites with large multimedia contents that were unreachable using the existing domestic dialup lines, store them on permanent medias (such as CD-Rom), and view them offline. In general, we needed a tool to collect very specific information for end-users from the WWW.

The HTTrack project was born to fulfill these needs: an easy-to-use tool that would allow regular users to make copies of small – but important – parts of the World Wide Web. Its design was rather experimental: Internet and related network architecture were fairly new domains we were discovering – and in particular, website copying was a totally new subject for us. The experience acquired through the development of this project will illustrate the method – "the Art" – of copying websites, and the suggested solutions for the multiple drawbacks encountered.

4.2 The Parser

4.2.1 The HTML Core Parser

The HTML parser is one of the two core components in a Web copying tool. Given an HTML[1] page data – that is, essentially an 8-bit[2] text file composed of plain text and markup tags – and its associated information such as the original URL,[3] the HTML parser's goal is to scan the page to collect links, analyze them and pass them back to the crawler. The HTML structure is not relevant to collect links: we are primarily interested in a limited number of tags, such as "a" or "img" elements, that will potentially contain hyperlinks to other resources (images, style sheets, HTML pages, etc.). Their position inside the page is generally not important, except in specialized domains where advanced heuristics can attach additional information such as the theme being discussed "around" these tags, allowing to follow certain pages and not other (irrelevant) ones. For a regular parser, the only useful information is the tags and their embedded properties.

The simplified core automaton is fairly easy to understand: a linear scan of the HTML page data bytes, starting from the beginning, detecting starting tags (<) and recognizing the various HTML elements by their names (see Fig. 4.1).

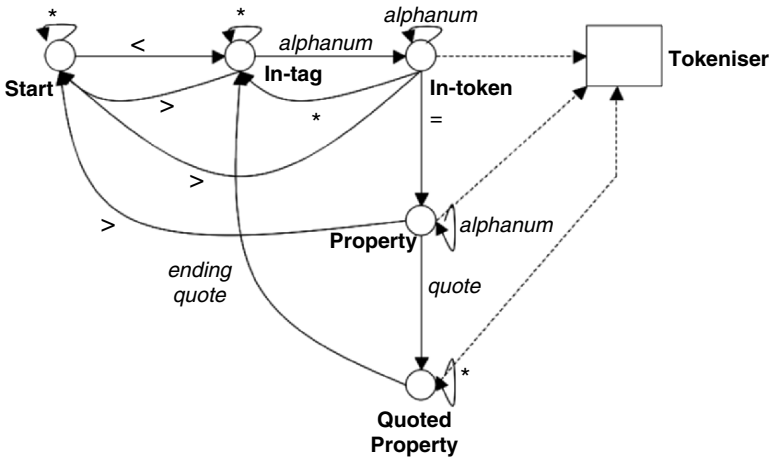

Fig. 4.1. Core parsing automaton

[1] See [1866].

[2] Note that the page character encoding will be important for link naming, especially on UCS2 file systems (including Windows ones).

[3] See [1738].

There are two classes of items to recognize inside HTML tags: tag names, such as “img” or “a”, and tag properties, such as “href ” or “src”. We can split these tags in two main groups: tags that allow to embed resources (such as images or style sheets loaded in the current page), and tags that allow to navigate to other resources (hyperlinks). For a given page, you can skip irrelevant links from the second group (e.g., links beyond the scope of the mirror) – the links will be unreachable in a disconnected (offline) environment, but this will not change the page aspect. But you have to be more careful concerning the first tag group, or the page will not be properly viewable when disconnected. In particular, you may have missing images or a totally broken page layout due to missing elements, such as style sheets or embedded scripting files. Hence, link URLs are not the only information that will have to be passed back to the crawler: the “tag context”, such as whether it is an “embedded” resource or not, will also be important to take the decision “take this link or not.”

Links themselves will be extracted by the tokenizer by analyzing well-known properties, which will be converted in their absolute form[4] using the original page URL: a protocol part, “`http:`,” a host authority “`//www.example.com`,” and a relative path “`/index.html`.” For example, the relative[5] link “`/top.html`” inside the page “`http://www.example.com/foo/index.html`” will be converted into the link “`http://www.example.com/foo/top.html`.” The link position will then be checked to ensure that it fits the default scope of the mirror; the check consists of a regular expression which value is by default the main URL prefix. If we started the mirror from “`http://www.example.com/foo/index.html`,” the default scope would be “`http://www.example.com/foo/*`” in a pseudo-regular expression syntax. Hence, links such as “`http://www.example.com/foo/top.html`” would be included in the mirror by default. Of course, additional rules might be necessary depending on the site being mirrored: the default expression shall then be customizable. At last, duplicate links must not be transmitted to the crawler twice: the parser has to keep the state of all known URLs and avoid retaking multiple times links that were already taken.

The parser also has to handle very numerous syntaxes, which can mix relative or absolute URL forms, HTML escaping[6] (such as ` `), URL escaping[7] (such as `%3a`), and in general a loose syntax. This syntax tolerance

[4] See [2396], Sect 1.4 “Hierarchical URI and Relative Forms”
[5] See [1808].
[6] See [1866], Sect 14. “Proposed Entities.”
[7] See [1630].

from browsers is high: even with really broken pages (including errors in tag syntax), browser will generally do their bests to pursue its analysis, rendering "what could be understood."

As an example, the absolute link form:

"`http://www.example.com/page 2.html`" can be referenced using multiple syntaxes, including several incorrect or unadvised ones. In any ways, the URL has to be recognized and took in account as the browser would have done.

<a href= "page 2.html"> (*double quoted link)*

<a href= 'page 2.html'> (*single quoted link)*

<a href= "page 2.html"> (*HTML-escaped characters)*

`<a href= "page%202.html">` (*URL-escaped characters)*

`<a href= page%202.html>` (URL-escaped characters, no quote)

`<a href= page%202&#46;html>` (multiple-escaped characters, no quotes)

`<a href= "page 2` [CR]`.html">` (*unexpected carriage return)*

`<a href= "http:page 2.html">` (protocol in URL, but no host)

<a href= "//www.example.com/page 2.html"> *(no protocol scheme)*

<a href <a href= "page2 [CR] .html"> (*broken tag syntax)*

At last, links have to be rewritten to fit the mirrored website structure. Links using the absolute form, such as "`http://www.example.com/index.html`", needs to be converted into relative form, such as "`index.html`" ; and links beyond the mirror scope (links that did not match the default regular expression scope) have to be rewritten in their absolute form. Hence, mirrored pages needs to be modified to be useable in a local structure.

4.2.2 The Script Parser

Several months after the beginning of the HTTrack development, and despite of improvements in the HTML parser, there were a fairly numerous websites that were not correctly copied, with a lots of missing images, missing files, causing navigation errors, because the parser just did not "see" these links.

Inside HTML pages, specific scripting[8] zones must be considered, such as JavaScript (active code inserted in pages), which demand specific parsing. Unlike HTML tags, which are objects rather easy to analyse, script code is nearly impossible to fully handle: the logic behind variables, functions and expressions can potentially be unreachable. First, even with a

[8] See the ECMAScript scripting generalization, [ECMA-262].

complete JavaScript interpreter, actions triggered by mouse position, clicks on elements, or environment (the time, client variables and in general, environment entropy…) can not be captured. Second, the capture of links using an interpreter would not solve the other problem: modifying the code logic to "fit" the mirrored site. Detecting links is not sufficient: we also have to modify them. And if this is a fairly easy thing to do inside HTML tags, doing the same inside complex scripting code is nearly impossible.

Hopefully, in most cases, the JavaScript code used is simple enough to fit the limited analysis abilities of a program. To dynamically load images – or to cache them in the background, Web designers generally use direct assignment to object properties using static strings, such as "`foo.src=bar.gif`" or, to open new windows, use expressions such as "`window.open("foo.html")`." A rough 80% of links hidden inside script zones can be detected – and modified – by handling these simple cases. The remaining cases, using expressions or unknown methods, will just be left as it. The result will not be perfect, and – concerning HTTrack – we knew from the beginning that everything will not be. The objective was to reach an acceptable quality level, that would allow to take care of most sites. With similar algorithms, CSS (style sheets) zones can be parsed with an acceptable quality.

The following simplified automaton describes the text strings extraction inside scripting areas (such as <script> tag sections). After extraction, the string analyser attempts to guess whether the data appears to be a link or not, depending on its form: strings terminating with a known extension, such as "`.gif`" or "`.html`," or strings starting with a protocol part such as "`http:`."(see Fig. 4.2).

Additional heuristics are necessary to limit detection errors in the string analyser, and especially the characters preceding the string: substrings inside expressions must be avoided, such as in these two examples.

foo.src = "/images/welcome.gif";
bar.load("/docs/welcome.html");

We can assume that "/images/welcome.gif" and "/docs/welcome.html" refers to the current document's location – this simple heuristic is safe in most cases.

foo.src = dir + "/images/welcome.gif";

No assumption can be made for "/images/welcome.gif": the preceding character "+" clearly shows that the string is part of a string expression, and therefore the complete URL cannot be guessed without language analysis.

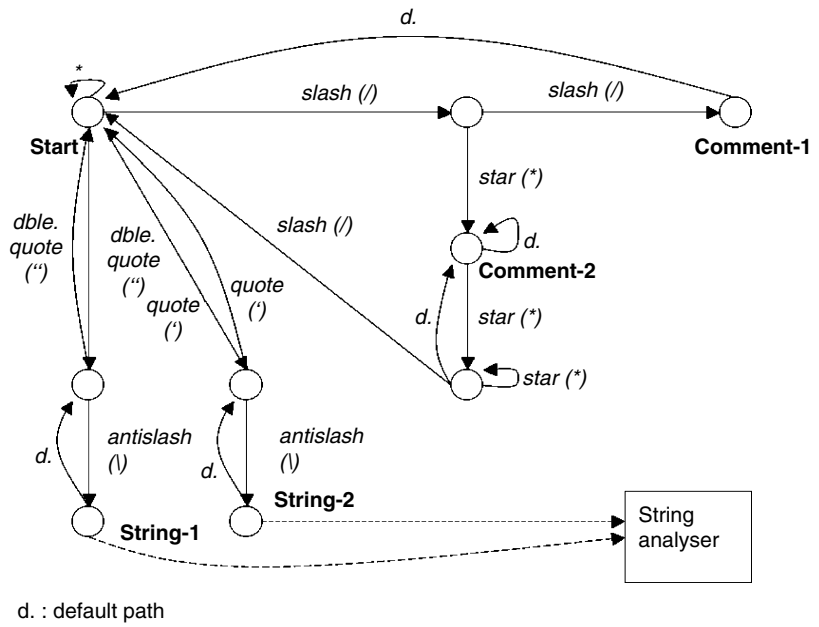

Fig. 4.2. Text strings extraction in scripting areas

Safe preceding characters includes “,”, “(“, “=”. Safe following characters includes “;”, “,” and “)”.

At last, strings selected as “probable URLs” must be modified to fit the local copy, as we do inside the HTML tags.

Having completed the automaton and the string analyser, we are able to handle most JavaScript cases. Greatly improving this algorithm is unfortunately very difficult, and would probably involve function and expression analysis, based on javascript language specification, and other advanced heuristics that are gar beyond the scope of a generic parser.

4.2.3 The Java Classes Parser

Another challenge is the handling of Java applets, which were quite popular at these times. After a while, the Microsoft-centric activeX format emerged, until waves of Trojans and viruses exploited this new format. Nowadays, the trend is more to use Flash applets, especially to design hideous and irritating advertisements you can not block. But in any cases, the annoyance for Web archivists remains exactly the same.

Let it be clear: if embedded scripting is annoying, binary embedded files such as java or Flash are a real pain in the neck. With text-based files format such as HTML or XML, you can easily modify subparts of the file without bothering about other ones. You can change a description in a far level; it will not impact the whole file. You can disregard most elements without even knowing their meaning. To summarize, you can focus on very specific elements – such as links – modify them, and forget the thousands other ones. Because these formats are intended to be simple, flexible, and easily understandable by human beings, and you do not need to understand the thousand-pages specification to get one single information. But binary formats such as Java classes rely on complex structures[9] that cannot be modified easily. Dare you modify a single string inside the file, and the whole file can be spoiled, because the string size, location and possibly content may have been referenced elsewhere using a collection of obscure pointers, offsets, checksums, and other magic tricks. For each binary format, you have to implement some complex algorithms to disassemble the desired data. Modifying these data and reassembling the whole thing is even more complicated.

A reasonable strategy to handle java classes is similar to the JavaScript heuristic, able to handle simple cases, not want a perfect binary Java parser. Hence, a basic class analysis based on embedded strings inside the .class file data segment is sufficient to enumerate interesting data (e.g., URLs). Here again, strings that "looks like" URLs will be kept. But the main difference is that we will not make any changes inside the binary file due to the complexity of this task. Besides, embedded strings representing relative links would probably work on a local environment, but not fully-qualified ones: modifying the strings without understanding (or analyzing) the logics behind would not be sufficient anyway.

Scanning interesting strings inside a Java .class file is not very complicated: we first have to (down)load the class file header in memory (10 bytes) and check the 32-bit integer magic ("0xcafebabe") to ensure that the file is explicitly a java class. See below the .class file header. The constants pool includes all static strings, such as strings used for URLs, and indexes of class name strings used to include specific java classes.

```
ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count-1];
```

[9] See "The Java Virtual Machine Specification – The class File Format."

```
        u2 access_flags;
        u2 this_class;
        u2 super_class;
        u2 interfaces_count;
        u2 interfaces[interfaces_count];
        u2 fields_count;
        field_info fields[fields_count];
        u2 methods_count;
        method_info methods[methods_count];
        u2 attributes_count;
        attribute_info attributes[attributes_count];
    }
```

Then, continuing to following the JVM Specifications,[10] enumerate all constant objects (`constant_pool` structures), and analyse all strings (`CONSTANT_String` and `CONSTANT_Utf8` objects) inside the file. Imported java class names can be detected using their corresponding objects (`CONSTANT_Class_info` structures): all specific (i.e., not standard library classes located in the java/ section) imported libraries can be retrieved later, as embedded links inside HTML data would be retrieved.

See Table 4.1: the "interesting" objects to scan inside a .class file are static strings that can potentially represent URLs, and Classes string indexes used to import external libraries (classes).

All detected links are then transmitted to the crawler, and the java class is stored untouched.

Despite a relatively high number of bogus mirrors, the result is rather positive: reasonably simple applets generally work, including applets composed

Table 4.1. Objects to scan inside a class file

Constant type	Value
CONSTANT_Class	7
CONSTANT_Fieldref	9
CONSTANT_Methodref	10
CONSTANT_InterfaceMethodref	11
CONSTANT_String	8
CONSTANT_Integer	3
CONSTANT_Float	4
CONSTANT_Long	5
CONSTANT_Double	6
CONSTANT_NameAndType	12
CONSTANT_Utf8	1

[10] See "The Java Virtual Machine Specification – The class File Format."

of multiple classes. But more complex applications generally fails due to missing files; either because the link was not modified inside the class, or because the link was invisible to the parser due to the hidden logics behind the binary code.

4.3 Fetching Document

The other engine element in a copying tool architecture is the robot responsible for gathering data from online websites – HTML pages, images, style sheets, and in general any media file available on the server. A stack of URLs to be collected, initially filled with one or more "root" addresses of HTML pages given by the user, is used by the robot which connects to respective servers, sending requests, and handling downloads. This robot can work in parallel of the parser to improve performances: while the parser is scanning pages, the crawler downloads data using multiple connections, dispatching ready files to the parser.

The crawler/parser interactions can be summarized in the following diagram: these two processes share a common bucket of links – the heap – filled by the parser as new links are being discovered, and emptied by the crawler as new links are being successfully downloaded (see Fig. 4.3). You can see it like a perforated bath tub: the number of remaining links to be downloaded varies with the time, to reach the count zero when the website is completed. At this point, the mirror is considered finished. Note that the parser only scans files known to contain links: HTML pages and other limited formats (such as Java or Flash files). Other files (such as images) will be stored "as is" on disk, without modifying them.

To fetch files, the crawler first fetch an URL in the link heap, and decompose it in three components: the protocol (http, https, etc.), the authority (the Web server hostname), and the path (including the query string – the optional part before the "?"). Note that the fragment (the optional part after the "#" character) is not part of the URI, and therefore never included when collecting URLs.

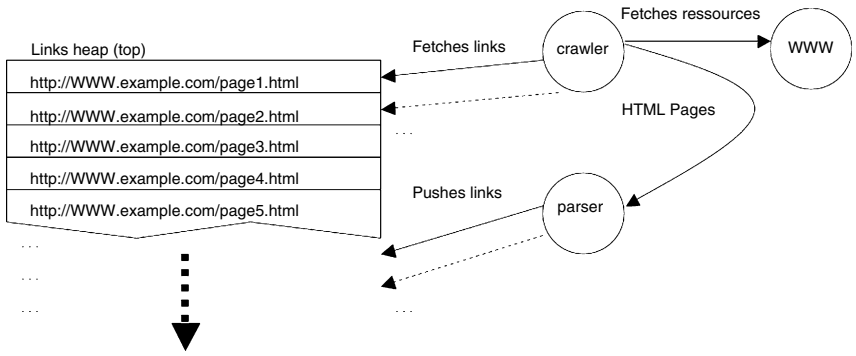

Fig 4.3. Crawler/parser interactions

The protocol will be used to dispatch the main data collecting routine: HTTP or HTTPS (HTTP with an additional SSL[11] layer), optionally ftp[12] or even the file[13] pseudoprotocol. Most links in the WWW will use the HTTP protocol, which is the only part we will discuss here.

When fetching an HTTP resource, four steps will potentially consume time or create latency:

- Hostname DNS resolution (wait for the DNS server response);
- Connection to the website (three-steps TCP connection handshake);
- Request sending (not really significant: less than half a Kilobyte, that is, one TCP packet – the ping latency will be more important here);
- Response fetching (depending on the transfer rate).

These steps can be processed sequentially for all links to be fetched. Or they can be optimized for most of them:

DNS resolution can be cached in the crawler so that further requests can be fulfilled without delays

Network latency created by connections establishments can be greatly reduced by using HTTP 1.1 Keep-Alive connections and a pool of connected sockets managed by the crawler

Response transfer rate can be boosted by using HTTP compression, especially for HTML data

[11] See [2246].
[12] See [959].
[13] See [1738], Sect. 3.10.

At last, specific HTTP handling such as the ability to continue the download of an interrupted file, can improve the overall performances on sites suffering from network instabilities

4.3.1 Authentication, Session, and Endless Loops

Many websites are not friendly to crawlers (and generally, nor to search engines) either because their hypertext pages uses technologies hostile to crawlers, such as Java/JavaScript or Flash, or because of their internal navigation scheme. Authentication or sessions are a first difficulty. Session-driven websites generally attach a "ticket" to each visitor when accessing the homepage, under the form of a cookie, an opaque data transmitted by the server to be used consecutively by the client for further HTTP requests. Authentication schemes either use username and password credentials, cookies, or both of them. Failing to pass the expected ticket can prevent you from accessing pages beyond the main entrance. Hence, the handling of cookies[14] or authentication[15] is not an option, nor the secure HTTP version, HTTPS,[16] which is the preferred way of securing the authentication process by avoiding any clear text credentials exchange. But cookies can also be replaced by identification elements included in all URLs as query-string parameters, or directly as part of the path segment. Entering of credentials can be expected through forms, not well suited for automated crawlers.

Each of these difficulties are annoying issue, as the website will probably attribute different identification elements each time the site is being updated, possibly rendering update impossible. Certain session-based navigation systems may also reattribute different tickets time to time after an arbitrary expiration time was reached, causing the robot to loose its way inside the site, and pushing it in endless loops and infinite mirroring. Other sites, especially those hosting forums or message boards, are using multiple different URL syntaxes to reference a unique page: duplicate files for each messages tend to dramatically increase the overall size being downloaded.

An attentive analysis of such sites, use of specific scan rules, and multiple tries will generally allow to overstep these problems. But it means manual, nonautomatic adaptations: fully automatic copies are sometimes beyond the capacity of mirroring tools.

---

[14] See "Persistent client state HTTP cookies" and [2965].
[15] See [2617].
[16] See [2818].

4.3.2 Redirections, Refreshes, and Frames

Both HTTP protocol and HTML specifications offers different ways to redirect a browser to another location, automatically: the browser first goes to the desired location, and is then automatically redirected to the new address without any user interaction. The first – and cleanest – method is to use a specific HTTP response (HTTP error codes 301, 302, 307), which will not return the final document, but instead it is new location. Websites moved to another place will still be accessible using their former address; which will gently redirect to the new one. This feature can also be implemented using specific HTML tags, metatags, that can replace certain HTTP headers such as the media type (and, for HTML pages, the character encoding being used) or the status code itself, using the "refresh" feature. At last, invisible frames can be used to "embed" pages located in another place. All these tricks are generally invisible to the user. But not to crawlers, that needs to take a decision: must the various redirections be followed, or not? Following by default redirections can be useful, if the site was moved elsewhere. But it can also be dangerous, it the site uses this technique to generate a list of external links that can be logged by the server (for popularity statistics), or if the site uses redirects to the main page instead of "404 errors" – a bad habit used by many free hosting providers. For this reason, the website copier HTTrack is not following by default redirects – except if the redirected place is allowed by the scan rules given by the user.

4.3.3 Connectivity

When used in corporate environments, Web crawlers face network restriction problems, such as the inability to access the Internet directly. Handling of proxies (delegating server used to fetch remote resources) is then an appreciated feature. Proxies are also used as protection and to accelerate Web responses inside internal networks, allowing to get a fast access to pages already visited. This interesting feature was for example exploited by administrators who used HTTrack to pre-cache external websites before presentations, providing better response times during the show as the proxy already had all the pages in cache! HTTPS is also a nice feature to be implemented for corporate environments, and not only for authentication issues: many corporate sites are only accessible through a secured connection. At last, implementing the next-generation protocols such as ipV6 is not an option: despite its currently limited usage range, it should be progressively widely used.

4.3.4 Politeness: Bandwidth and CPU Limits

Implementing bandwidth limiters and limiting the number of connections is clearly an important security feature for a WWW crawler. This protection is specially important with nowadays lines, as Internet Providers can now offer several megabits of bandwidth in urban areas even for domestic lines. A regular user can potentially cause denial of services when downloading a website at "full speed." Abusing the bandwidth would not only cause legitimate complaints – but would also lead to ban mirroring tools everywhere, ruining all archivists efforts.

As an example, even in the beginning of the HTTrack's development, bandwidth and CPU usage were important issues. The bandwidth available for testing and validating the crawler was fairly high in our development environment, allowing to run multiple crawls in parallel or reissue regular benchmarks to fix bugs, detect new problems, and improve the engine in general. It was rather easy to get very fast transfer rates – too fast for many servers, which could be quickly overloaded.

This remark is also true when accessing resources that require processing time on the remote server, notably when fetching dynamically generated pages with underlying costly processes such as database operations. Using multiple connections to access these pages will lead to overload the machine serving the data, causing another kind of denial of service.

4.3.4.1 Politeness: robots.txt

In many cases, websites containing area crawlers should not automatically process: large files, costly dynamic pages, sections with potentially endless loops, and in general parts that are not suitable for fully automated processes.

For these reasons, webmasters now places specific hints for robots that can be found on the server in a file called "robots.txt." This simple ASCII file lists areas that crawlers should or should not visit for various reasons, using a simple syntax, optionally specifying names of crawlers concerned by each rules. The robots.txt[17] standard is an important feature to implement when designing search-oriented crawlers, but in certain cases, archiving crawlers will have to bypass them when areas not suitable for indexing – but suitable for archiving – have to be processed. In such cases, Webmaster's cooperation is highly recommended, especially to suggest which areas can be safely crawled.

[17] http://www.robotstxt.org/wc/exclusion.html

Example of robots.txt rule explicitly forbidding the/private subarea from crawling:

*User-agent: **
Disallow: /private

4.4 Create an Autonomous, Navigable Copy

From the beginning, an implicit choice was made for the crawler: as a regular (human) user would, copied files are stored in a filesystem (i.e., on a disk as regular files) rather than inside a database, for example. The result would be directly viewable with any browser, as if you were accessible the "real" website. This choice is not only an intuitive choice. It is also a security choice. A database-driven copy would require the corresponding database software to be useable, not necessarily compatible with all operating systems and architectures. This software will possibly be deprecated in several years, and impossible to run on future systems. An archive that might be unusable in the future because of a software component would be a grave design choice for preservation purpose. Similarly, any nonstandard software that might be necessary to view the copied website would fall in the same critical problems. Copying Web resources into regular files that would be easy to backup, mapping their names as regular filenames, and allowing any standard (i.e., respecting Internet standards) browser to access them is an obvious and reasonable choice. Being independent from any application vendor, operating system vendor and even machine architectures is one of the keys of the Web preservation.

Another options to use standard container files like WARC but at the time we started developing HTTrack, this standard did not exist.

Copying live sources such as HTTP files to local file systems and browsing them without the need of any additional programs but a regular browser has several drawbacks. The first problem lies in the way you organize the file structure locally. Online resources rely on URLs which, unlike filesystem paths, follow arbitrary naming convention that depend on the remote server implementation. A Unix fully qualified path "/home/users/smith/document.tex," or a Windows fully qualified path "C:\Documents and Settings\smith\My Documents\document.doc" both follow strict conventions, such as characters allowed in filenames and directory names, restrictions on their size, and a specific character (/or\) playing the role of a separator. In the contrary, Web Servers can choose to follow a traditional structure, such as

`http://www.example.com/foo/bar.html`, or in the contrary use exotic naming conventions, such as `http://www.example.com/foo; bar; t=html;q=1//`. Luckily, most servers are gentle enough to avoid such eccentricity – hence we can consider that in most cases, URLs will look more or less like regular paths, allowing to recreate a similar filesystem structure when mirroring a site. The URL "`http://www.example.com/foo/bar.html`" will then be copied as "bar.html" somewhere in a subdirectory "/foo/bar/." However, many links do not have any document name, and use the default "/" convention. The link "http://www.example.com/foo/bar/" can be stored in the "/foo/bar" subdirectory – but it also needs a filename, such as "index.html" or "default.html." One of these arbitrary names will then be choosen. But what to do if both "http://www.example.com/foo/bar/" and "http://www.example.com/foo/ bar/index.html" exists in the website we are copying? Similarly, a number of forbidden characters must be avoided because they are forbidden or discouraged on Windows[18]/Unix[19] environment, and additional restrictions must be taken in account if the website is to be stored on CDRom media[20]. At last, Windows filesystems can handle Unicode characters in filenames such as accents; characters that can not always be represented easily on all platforms. In all these cases, the offending characters can be changed by replacement characters such as "_" (underscore), and names shall be truncated if necessary – but other strategies could be used, such as escaping these characters using an arbitrary convention like URL-escaping.

Another scenario has to be handled, too. When opening a document on a local disc, the file extension, such as "pdf" or "html" also describes its type: an Acrobat document, or an HTML file. If you rename an HTML document as "pdf," you will not be able to open it anymore: the wrong program will be launched to view it, and will fail to do so. On a Web environment, media types are transmitted by the server to the client through HTTP headers: the URL naming scheme is not important anymore. The resource "`http://www.example.com/index_1.cgi`" can be an HTML document, a PDF document, an image, etc. it is the media type transmitted by the server when fetching the document that will be decisive. We have to rename the resulting document if we want it to be useable in a non-Web environment: index_1.html, index_1.pdf or index_1.jpg depending on the media given by the server.

---

[18] The characters / : * ? " < > | are forbidden, among with other restrictions (case insensitivity , reserved names…).

[19] The use of ~ | * is discouraged for shell-expansion reasons.

[20] See ISO9660 restrictions, notably filename size limits (30 characters) and forbidden characters * / : ; ? \.

Here again, the consequences of these adjustments have to be considered, as different URLs can have identical local names after applying naming restrictions. Consider the following four URLs pointing to an HTML document:

http://www.example.com/index_1.html
http://www.example.com/INDEX_1.HTML
http://www.example.com/index:1.html
http://www.example.com/index_1.cgi

All these links would get the same filename, "index_1.html," as the second URL is identical to the first one except for the case (but Windows environment are case insensitive), the third one contains the unadvised character ":," replaced by "_," and the fourth one will be renamed "html" instead of "cgi" to be viewable offline in a filesystem. Avoiding such collisions means that different names must be found when they occur – like "`index_1.html`," "`index_1-2.html`," and "`index_1-3.html`," and require adjustment in the parser which task will also be to apply these changes in downloaded HTML pages so that links can refer to the respective local files.

4.5 Handling Updates

4.5.1 Updating the Copy

One drawback in making static copies of websites is that these copies will not change anymore, they will not be updated by their original authors, and as books in libraries hold their typing errors and mistakes, copies remains exactly the same as they were during their creation. They may deprecate. Depending on the archivist's needs, it might be desirable to make regular copies of preserved websites either to and get up-to-date version, or to regularly store a copy that would allow to retrieve the site on a specific moment.

One solution is to retrieve the entire website when necessary, repeating exactly the same operations. But when making regular copies of websites, updating (i.e., only transferring modified content) tend to be a major issue, especially with big websites containing or with large media files. Recrawling exhaustively a website in the purpose of having an up-to-date copy is a very inefficient method: waste of bandwidth, waste of time, and waste of storage space when handling multiple site versions. A regular browser usually stores recently pages and associated files in a specific location

called “cache,” which can be used to avoid retransferring data that was previously consulted. When visiting pages already in cache, the browser is able to ask the remote server whether its local copy is fresh or if it has to be transferred again. Here again, the solution for updating problems is to mimic browsers, by handling a cache that will be used by the crawler to check the freshness of already downloaded data.

There are two main mechanisms described in HTTP 1.0[21] and HTTP 1.1.[22] The first and most widely used one is the old HTTP update mechanism, which rely on the remote document date to ensure that the resource is always up-to-date. By sending the document’s date to the client, the server allows it to perform further update check by asking something like “Please give me the /index.html document, except if it was modified since 14 July 2002.” The second mechanism is a more general system, which uses an opaque string aimed to identify a specific resource content. It can be the document’s md5 checksum, for example, or any other element that can be used by the server to identify the document’s version/freshness: it can also be the document’s last-modified date, but this is only a particular case.

The diagram above gives a rough idea of a caching mechanism (see Fig. 4.4). When fetching a link, the crawler first checks whether it is already known by the local cache. If not, the file is downloaded as usual. But if the cache already has a version, the crawler can either decide to ask the server for freshness, or to directly take the existing cached file – when recovering an interrupted or crashed or mirror session, for example.

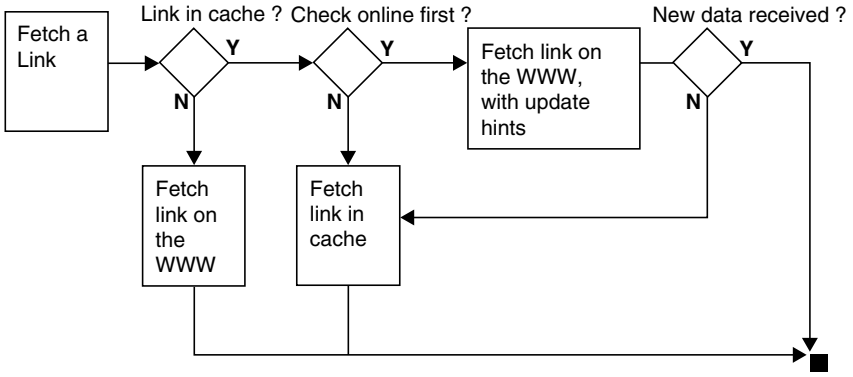

Fig. 4.4. Caching mechanism

[21] See [1945], Sect. 10.9.
[22] See [2616], Sect. 14.19.

The cache can store files such as images or binary data, or reference their location if they already exists on the mirror file tree: in this case, original HTML data needs to be stored "as is" anyway in the cache, because the existing files which were modified by the parser and can no longer be used effectively due to URL mangling (you can not guess the original URL because of that). Additionally, Web server metadata must be stored among file data such as original media type, status code, and of course information needed for updating purpose: the "Last-Modified" date, and/or the "Etag" opaque string. Note that the cache files are not necessary to view the mirrored website, but only to update it, and thus can be omitted when making navigable copies such as CD-Rom extractions.

4.5.2 Storing the Updated Copy

Being able to update a copy is one thing. Propagating the update in the copied archive is another one. On a filesystem tree, updated files can replace previous versions (overwriting them) and new HTTP header information can be merged in the cache. But updated files can also be handled using a file system managing versioning, such as a CVS[23] tree, allowing to crawl dated copies without needing to store multiple copies of the same files. If control version directories are too difficult to implement because of the necessary versioning interface, or are not desirable because is ties up the archive to a specific software, an intermediate solution is to use native file system's linking (on Unix systems, symbolic or hard links) and organize versions in different directory trees, with files or directories either pointing to the preceding copy, or storing the new version depending on the website changes.

On this example, the first copy made in January is 50 MB large. The update, issued in February, modified an overall of 3 MB of data. The remaining files (47 MB) were untouched since January. The third run modified again 2 MB of data, and brought 2 MB of new files (4 MB of new material overall), with 46 MB of remaining files untouched since January, and 2 MB of files untouched since the February update. In this imaginary example, the three sites versions represent a total of 152 MB of virtual data, for a total of 57 MB of physical data (see Fig. 4.5).

In many cases, updates are far less expensive than the "first run": handling versioning is generally an inexpensive feature compared to multiple identical copies.

[23] Control Version System.

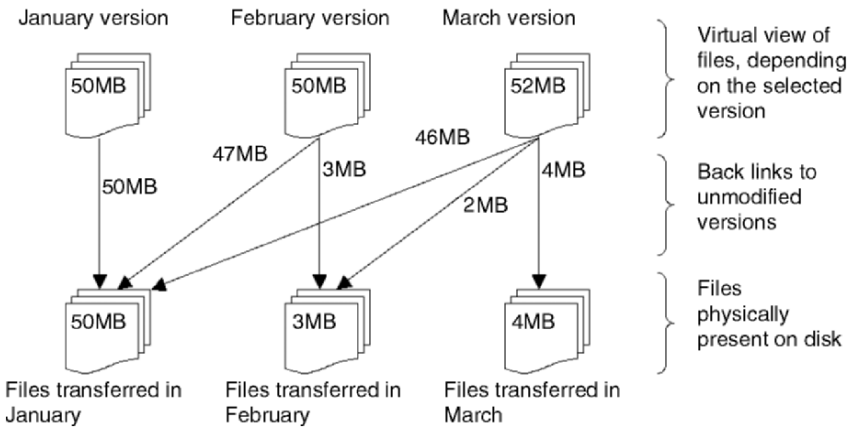

Fig. 4.5. A website copy with two updates

4.6 Conclusion

Preserving and archiving websites is a thrilling technical challenge which must be pursued, endlessly. Evolving technologies, evolving contents and evolving growth of the WWW means that no definitive archiving solution might ever be found. Existing solutions can still be improved as no perfect system was yet made. This continuous work in improving Web preservation techniques is done by passionate people around the world, in libraries, universities, companies and by individuals.

Reference

The WWW was built over Internet standards, especially through the Request For Comments (RFC), which describes most protocols and standards when using Internet technologies. They are public, free, and easily understandable for "regular computer programmers," unlike many other international standards. This easy access was one of the reasons why the entire Internet grew so quickly: a common standardized technical base that everybody (i.e., not only accredited companies) could use.

Below you will find several RFC highly recommended when developing WWW preservation tools. This list is not exhaustive, and shall be completed by references indicated at the end of the documents described below.

HTTP References:

– RFC 1945 – Hypertext Transfer Protocol – HTTP/1.0
(first version of the HTTP protocol)
http://www.ietf.org/rfc/rfc1945.txt?number=1945
– RFC 2616 – Hypertext Transfer Protocol – HTTP/1.1
(second version of the HTTP protocol)
http://www.ietf.org/rfc/rfc2616.txt?number=2616
– RFC 2818 – HTTP Over TLS
(also called "https")
http://www.ietf.org/rfc/rfc2818.txt?number=2818
– RFC 2660 – The Secure HyperText Transfer Protocol
(not to be mixed with https)
http://www.ietf.org/rfc/rfc2660.txt?number=2660
– PERSISTENT CLIENT STATE HTTP COOKIES
(browser "cookies")
http://wp.netscape.com/newsref/std/cookie_spec.html
– RFC 2965 – HTTP State Management Mechanism
http://www.ietf.org/rfc/rfc2965.txt?number=2965
– RFC 2617 – HTTP Authentication: Basic and Digest Access
Authentication (used when authenticating to a server expecting credentials)
http://www.ietf.org/rfc/rfc2617.txt?number=2617

URL/URI References:

– RFC 1630 – Universal Resource Identifiers in WWW
http://www.ietf.org/rfc/rfc1630.txt?number=1630
– RFC 1738 – Uniform Resource Locators (URL)
http://www.ietf.org/rfc/rfc1738.txt?number=1738
– RFC 1808 – Relative Uniform Resource Locators
http://www.ietf.org/rfc/rfc1808.txt?number=1808
– RFC 2396 – Uniform Resource Identifiers (URI): Generic Syntax
http://www.ietf.org/rfc/rfc2396.txt?number=2396

HTML, Java and Script References:

– RFC 1866 – Hypertext Markup Language – 2.0
http://www.ietf.org/rfc/rfc1866.txt?number=1866

– Standard ECMA-262 - ECMAScript Language Specification
(standard based on JavaScript and JScript)
http://www.ecma-international.org/publications/standards/Ecma262.htm
– The Java Virtual Machine Specification – The class File Format
(binary specification of class files)
– http://java.sun.com/docs/books/vmspec/2ndedition/html/ClassFile.doc.html
– The Java Language Specification
http://java.sun.com/docs/books/jls/second_edition/html/jTOC.doc.html

Other Internet Protocols References:

– RFC 2076 – Common Internet Message Headers
(notably headers used in HTTP headers)
http://www.ietf.org/rfc/rfc2076.txt?number=2076
– RFC 2822 – Internet Message Format
(base RFC for many Internet protocols)
http://www.ietf.org/rfc/rfc2822.txt?number=2822
– RFC 2045 – Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies
http://www.ietf.org/rfc/rfc2045.txt?number=2045
– RFC 1950, RFC 1951, RFC 1952 – Compressed Data Formats
(used in HTTP compression)
http://www.ietf.org/rfc/rfc1950.txt?number=1950
http://www.ietf.org/rfc/rfc1952.txt?number=1952
http://www.ietf.org/rfc/rfc1951.txt?number=1951
– RFC 2246 – The TLS Protocol (used in https)
http://www.ietf.org/rfc/rfc2246.txt?number=2246
– RFC 959 – FILE TRANSFER PROTOCOL (FTP)
http://www.ietf.org/rfc/rfc959.txt?number=959