A technical approach for the French Web Legal Deposit

THOMAS DRUGEON

Institut National de l'Audiovisuel
4, avenue de l'Europe
94366 Bry sur Marne cedex
France

thomas.drugeon@ina.fr

Abstract

In this paper we present the technical approach we developed at the National Audiovisual Institute (INA) for the forthcoming extension of the French legal deposit law to web contents. This paper covers crawling and storage aspects of the project. The INA being in charge of the archiving of websites which are related to the media and AV (Audiovisual), focused crawling strategies are used to discover and subsequently archive relevant websites. A two-level crawling architecture is described, carrying on focused and continuous crawling policies using the web site as the unit. Thereby, issues raised by the harvest of the internals of a website and those of managing the global crawl are handled independently, allowing for finer control and better comprehensiveness in data archiving.
The lower layer of the storage model used to archive and preserve web contents relies on a custom, distributed file system, using the same networking solutions as the crawling system. Specific issues led by streamed AV contents are taken into account in both crawling and storage parts.

Introduction

The French legal deposit is presently extending its scope to cover online public web contents. Being in charge of the radio and television legal deposit since 1992 (which was enforced from 1995 onwards), the INA will be responsible for the collection and preservation of French websites with relation to media and AV (Audiovisual) contents, as an extension to its current missions. In parallel, the French National Library (BNF) will take care of other French websites, with a particular interest in online writings and newspapers.
This criterion of media relatedness defines a coherent web portion inside the French web panorama. Early investigation has shown that this portion consisted in only ten thousand to fifteen thousand websites, but accounted for more than half of the entire French Web in terms of storage needs.

Furthermore, media websites are particularly difficult to handle, with technical specificities such as streaming and rich media. In fact, the web is becoming a major media, working with the same publication logic as broadcasting, as well as with specific forms of user-interaction. Being a point of convergence, the Web brings together issues from all existing sorts of media, and forces us to redefine the notion of archived document. Acts of publication and consultation become similar on the Web, leading to an interaction-based archiving system. In fact, while being mostly used as a broadcasting media, the Web remains unicast in its access mode. The archive process has to mimic user consultation to discover publications, emphasizing interaction issues with the many technical formats and communication protocols that constantly emerge on the Web.

Alongside, the definition of a web portion leads us to a delimitation process to discover and follow websites we are in charge of. In fact, there are no lists or authority on which we can rely for this delimitation, and focused crawling techniques have to be used. Experiments revealed that nearly 50% of media-related websites were hosted in the ".com" top level domain (TLD), and only 30% in the ".fr" one. More than two hundred thousand websites had to be crawled using focused crawling techniques in order to obtain a consistent corpus of ten thousand relevant websites[1].

In addition to having to maintain this spatial delimitation, the crawling process also needs to continuously harvest and archive websites in their temporal delimitation, i.e. update frequency. Actually,

[1] Relevance of website was automatically estimated during our experiments. Results might slightly differ when human judgment will enter this process.

a legal deposit implies a responsibility of exhaustiveness in archived data that is difficult to obtain on a media entirely built on "best effort" principles.

In this article, the architecture of the crawling system will be described in the first section, whereas the second section will discuss the low level storage architecture.

1 Crawling system

The INA crawler was designed with scalability in mind from the outset. It is tailored to be efficiently distributed across commodity PCs without synchronization overhead. It is also meant to be modular and flexible.

1.1 Flat architecture

Most existing crawling software such as Mercator [1], Polybot [2] or Heritrix [3] rely on what can be called a flat architecture crawling system: pages are taken from a global URL queue, downloaded and analyzed, and their links are appended back to the queue. Conceptually, this is a basic mechanism that can be extended from a website analyzer to a global distributed web crawler, but further thoughts force the design to become overly complex:

- The crawler needs to know which pages were already downloaded or are pending in the queue. Handling such data structures in an efficient manner gets more difficult when the number of pages exceeds main memory capacities.
- For efficiency, DNS lookups and robots.txt information need to be cached in order to avoid requesting them for each page.
- Web politeness calls for crawlers to interact gently with each website. This includes contradictory directives, such as limiting the number of pages fetched in a given period of time, while in the meantime keeping HTTP connections open for as many requests as possible. Thereby, the crawler needs to have different queues for different websites, as well as some other site-only information (such as DNS or robot rules raised above).
- When crawling significant portions of the web, the need for a distributed system becomes crucial. Synchronization between crawling processes (URL exchange, site-specific rules, etc.) becomes an issue.
- Site specifics crawling strategies such as robot traps avoidance or session handling have to be handled inside the global crawl.
- Focused crawls are hazardous to control because the decision of following links has to be taken for each downloaded page independently.
- Incremental crawls are difficult to deal with because the update of each page has to be scheduled independently.
- The crawler has to regularly save its state in order for the crawling campaign to be resumed in case of failure.

However appealing and efficient this kind of design might seem when used on a single machine, it tends to become cumbersome when applied to large-focused and continuous distributed crawls.

1.2 A two-level design

Instead of using a flat architecture, the INA crawler is built with two levels of architecture:

1. Site crawlers collect pages inside a website, deciding which links have to be considered internal to the site and followed. External links are listed and reported.
2. A scheduler collects websites in a web portion, deciding which website (reported by site crawlers as external links) has to be considered internal to the focused web portion and followed. Websites are rescheduled for continuous crawling.

Although these two layers are very similar in concept, each handles its own specific issues independently from the other.

The scheduler can be considered as a website-based focused crawler, whereas flat architecture crawlers are page-based. The complexity and specific constraints of harvesting a website are encapsulated inside site crawlers, without URL overlap between crawlers or synchronization issues, letting the scheduler concentrate on its own scheduling policy at the website level. This releases a great number of technical constraints against the scheduler, and an on-disk database system can be used without speed

penalty, warranting for large crawls to be handled without memory limitations. This also enables the scheduler to resume a crawl campaign at the exact point it stopped at in case of failure. On the contrary, site crawlers store their random-access data structures in memory and cannot resume an aborted crawl: if a failure occurs during a crawl, the website has to be entirely recrawled from its first page[2]. The scheduler is prepared to use another site crawler for the same task if one fails.

Each of these two levels in our architecture will be discussed in details in further sections.

1.3 Networking solutions

Building a highly efficient distributed crawling system requires specific developments in network area as well as data structures. Most components were written in Perl and C (C parts being embedded in Perl code) for flexibility. Flexibility is the main requisite for the crawling system, because we always need to adapt to new web technologies and issues.

1.3.1 Anet

Anet (Asynchronous Networking) is the Perl framework we developed to unify and optimize our design for all network-related software. It enables to concentrate on application logic instead of having to handle networking considerations

Previously developed software being developed in Perl (notably the site crawler), we had to use the same language for our framework. Unfortunately, the only existing Perl framework, namely POE [4], was not scalable and fast enough for our purpose. POE was also too intrusive, the whole application design having to be build using its logic, which renders its integration into existing software difficult.

Anet is event-based and manages multiple socket connections and time-based tasks using a single thread. It also manages multithreaded tasks. In fact, file system access functions being synchronized in most operation systems, they are handled using a pool of worker processes, triggering events inside Anet. This worker scheme can also be used to handle long calculations or other synchronized tasks such as RDBMS access.

Despite being mostly written in pure Perl[3], Anet achieves very good results by using the most appropriated network strategies and native system calls. In fact, it tries to use every possible optimization available on the operating system it runs on, in a transparent manner:

- The most scalable socket listener is chosen:
  - epoll for linux 2.6 and above
  - kqueue for FreeBSD and MacOS X (not tested)
  - poll for other Unix systems
  - select for MS Windows[4]
- ZeroCopy TCP is targeted:
  - sendfile is used on Linux, FreeBSD and HP-UX
  - TCP_CORK is used on Linux
  - TCP_NOPUSH is used on FreeBSD

In our tests, the performance overhead introduced by Perl was low, applications being mostly IO bounded.

Anet uses a specific callback system for all its event listeners. Callbacks can be code references, anonymous closures, or methods of an object. This flexibility makes Anet integration straightforward without implicating constraints in the global application design.

In addition to a low-level timer and socket event interface, Anet provides a high-level stream interface for file access and TCP connections, on top of which specific file format handlers and networking protocols can be implemented. Anet already includes several extensions, such as a full HTTP

[2] Such a function could be added, but we believe that the website is not the right level to handle crawl resuming.
[3] Low-level IO functions, the epoll wrapper, and the heap used to dispatch timers are the only parts written in C.
[4] Completion port was not implemented, Windows being merely only a test platform for us.

1.1 client/server implementation, a DNS client with cache, a Microsoft Streaming Protocol client (MMS), etc.

1.3.2 HIP

To connect crawling components, a specially designed asynchronous RPC (Remote Procedure Call) system called HIP (Hierarchical Interaction Protocol) was developed on top of Anet's TCP stream API.

Existing RPC systems like SOAP were not suitable for our purpose:

- In order for each agent to be able to initiate a request to a peer, each agent had to be set as both SOAP server and client. So each agent had to be reachable as a server. This renders geographical deployment difficult.
- SOAP libraries use a synchronous approach: the client calls a remote method and waits until it receive the result. This blocking behavior can be dealt with by using multiple threads, but this model doesn't fit well with our need of handling a large number of agents.

Every HIP agent can have one master to which it has to connect, and several slaves that will connect to it, forming a tree. HIP agents connect to their master via a TCP socket (tunneling can be used to connect through HTTP proxies via the 443 port, normally used for SSL connections). This single connection is kept open for the lifetime of the slave and is used to issue and receive requests and responses in both directions.

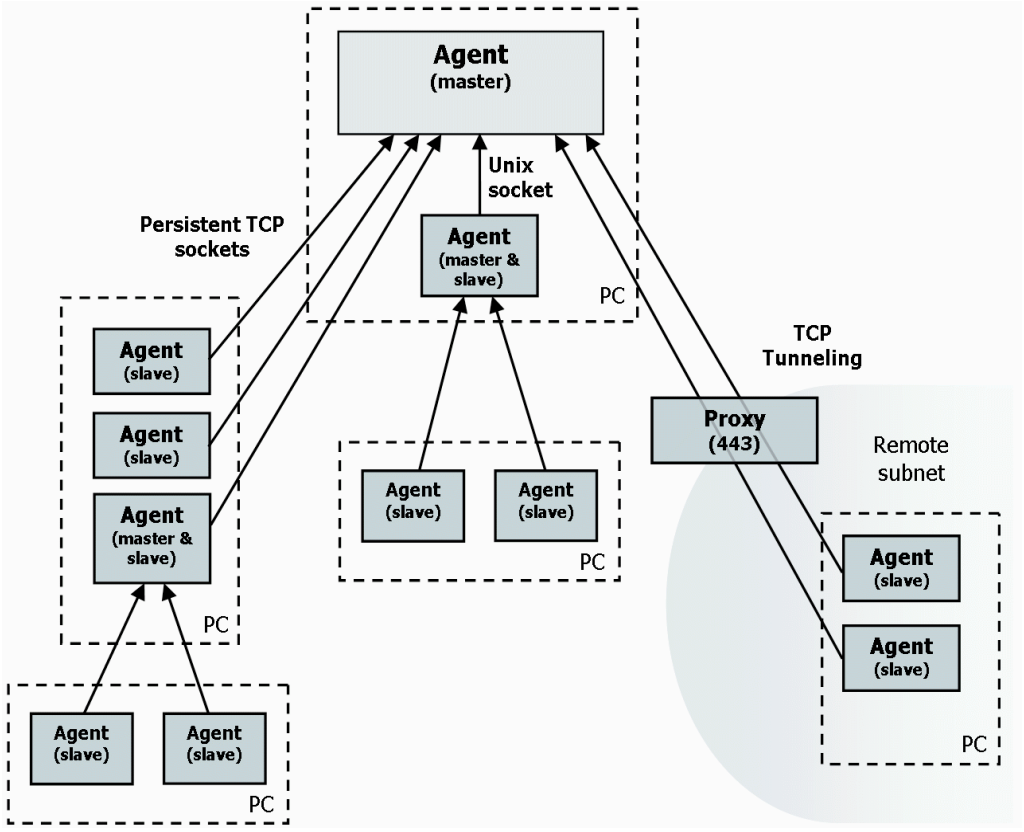

Figure 1: example of multilevel HIP network

Before exchanging requests and responses, the slave and the master have to agree on a common serialization protocol for the parameters of requests and the values of responses. When the same language is used on both nodes, the native serialization model is chosen[5]. If languages are different, YAML 1.0 [5] is used. Requests are not synchronized: when calling a remote procedure with HIP, the caller specifies an Anet callback to be triggered when it gets the response, thus avoiding the caller to block. Agents can emit a large number of requests to several slaves or to their master while still responding to their requests.

[5] Perl agents use the Storable module

Reponses can be received in a different order from the order of requests without causing trouble in callbacks triggering, even on the same connection.

HIP was meant to be a lightweight communication protocol. This goal is reached by using fast serialization mechanisms and persistent connections[6].

Although HIP was primarily developed for the crawling architecture, it is now also used for other parts of the project, such as the *replication layer* (cf. section 2.3).

In the crawling architecture, site crawlers are slaves of the scheduler. Site crawlers present a specified API to the scheduler to receive collect orders (among others). We can thus consider that site crawlers collect pages via HTTP whereas the scheduler collects websites via HIP.

1.4 Site crawlers

Site crawlers are HIP slaves, devoted to the scheduler as agents. Each site crawler can collect one website at a time, storing data locally, together with XML crawl analysis reports (external links, keywords, site graph, etc.). Websites that crawlers might encounter can be very different: some only contain ten static HTML pages with five GIF images whereas others include hundreds of thousands of dynamic pages with Javascripts, embedded FLASH, AV contents, etc. Site crawlers have to deal with all those websites efficiently, crawling small websites with a low overhead while being prepared to scale up to millions of complex contents without exhausting main memory or CPU capacities, or causing incessant disks access. In fact, disks access mostly only apply to buffered sequential read/writes (URL queues, reports, etc.), all random-access data structures being kept in main memory by using specially developed compact hashes and bloom filters implementations [6].

1.4.1 Crawling strategy

In order to collect a website, the site crawler needs to know its delimitation. The delimitation is defined by the scheduler in its collect order, and merely consists in a start URL, a maximum depth level and accept/reject expressions[7]. For each link it found in a page, the crawler will test if it matches the "accept" expression, and if it does not match the "reject" expression to determine if it is an internal or an external link. All links that pass the expressions are considered internal to the site and are followed, whereas other links are considered external and are set aside to be reported to the scheduler.

Example of delimitation for the TF1 website	
Start URL	http://www.tf1.fr/
Accept expression	//tf1.fr OR //tf1.com OR .tf1.fr OR .tf1.com
Reject expression	//eurosport.tf1.fr OR //eurosport.tf1.com
Maximum Depth	150

Links to internal pages are followed in a breadth-first manner up to the defined maximum depth or the end of the website.

In order to avoid robot traps or accidental infinite loops, dynamic contents are followed with a smaller *maximum depth*. Moreover, session IDs are detected using heuristics[8] and removed from URLs. The same heuristics are used when browsing the archive (cf. section 2.4.2).
To avoid recursive loops to inexistent URLs, the crawler also detects repetitive patterns in the path of the URL. The number of URLs refering to the same server directory with different files, or the same dynamic script with different parameters, is also restricted.

Going through an entire website can be quite long, all the more so when temporizations are used. However, the scheduler can decide to send a second site crawler on the same site without waiting for the first one to complete. The crawling being made in a breadth-first manner and with the same parameters,

[6] When using scalable notification mechanisms such as the ones that Anet uses, keeping a large number of connections open is a really lightweight task.
[7] These are simplified regular expressions
[8] URL parameters that correspond to 128bit or 160bit hexadecimal values are identified as session IDs.

the same pages will get downloaded with the same time interval without the need for synchronization between crawlers.

1.4.2 Copy exhaustiveness

The crawler maintains two types of queue for internal links it has to follow, depending on the way a real user would have encountered them:

- Links that would need a user action, such as a mouse click, are called *page links* and are appended to the next level queue (typical breadth-first scenario).
- Links that would have been automatically followed by the browser, such as images in a page or redirections, are called *component links* and are immediately followed (local depth-first scenario).

This ensures that components of a page will always be downloaded with the page they appeared in, may the maximum allowed depth be reached or forthcoming failures occur. It is important to note that the page/component distinction does not rely on the content type: an image is considered a page if it the result of a user's click (such as when enlarging a thumbnail image by clicking on it), whereas an HTML content is considered a component if it loaded from a frameset or a redirection.

A second concept used to typify links is the idea of requisites. A *requisite link* will be followed regardless of whether or not it fits within the delimitation of the website. *Requisite links* are thus always considered internal to the site. This notion is used to enforce downloading of images, scripts, and other elements of a page needed for its visualization, even if they are stored on other servers. There is a natural symmetry between compound/page and requisite notions. To wit:

- Components of a page are all requisites (images, scripts, embedded objects, etc.), except redirections, popup windows and frames.[9]
- Pages are never requisites, except the start URL of the website.

These two concepts are coupled to ensure that downloaded documents are fully consistent. Actually, being a tool of the legal deposit, the site crawler is required to be exhaustive and respectful of the data of the website. Documents cannot be modified in any way: even content normalization or link modification is to be considered outlawed. Likewise, omitting parts of a document, such as an image or a CCS style sheet, might alter its content and meaning noticeably. Thereby, particular care is taken not to miss any link.

1.4.3 Parsing the Web

Parsing is one of the most difficult issues to deal with when building a website crawler, especially when full exhaustiveness is a mandatory target. The crawling process is not intended to produce a mere index, but a real viewable copy: a single mislaid javascript link referring to a style sheet might alter the presentation of an entire website. Links have to be extracted from all sorts of contents, ranging from the usual HTML page to the very specialized embedded object. New content formats appear every year, some being proprietary and closed, some others being open-sourced and intended to become norms. We need to be able to handle those formats as they emerge as new standards. A plug-in architecture has been implemented, allowing for efficient integration of new content parsers. For further analysis, text of links and plain text of the page are also extracted by several parsers. It is the responsibility of the parser to define page/components links as well as require links. Links to streaming contents are also reported differently.

For efficiency, most parsers implement a chunk by chunk treatment and are able to parse a document in a reentrant manner, even one byte at a time, without the need to store the entire document in memory. Parsers that do not implement a chunk by chunk treatment have to accumulate a copy of the document in an in-memory bounded buffer before parsing it, and are thus only implemented for formats whose contents are typically small such as javascript or style sheets.

External programs are rarely used as parsers, and a core implementation is always favored when possible[10]. Each site crawler embeds the complete parsers collection[11].

---

[9] Being collected independently of their URL, the same requisite can be collected and archived by different crawlers for different websites (eg: a counter script).

[10] The only current notable exception is an external PDF handler, but a core implementation is underway.

[11] With the use of copy-on-write techniques for crawler's duplication, sharing parsers or loading them on demand would not be a benefit.

Current homegrown core-implemented plug-ins collection includes the following:

Plug-in	format	Links extraction	Link text extraction	Text extraction	Chunk by chunk parsing	Specificities
HTML	html	yes	yes (printed text and tool tips)	yes	yes	Embedded scripts and styles are extracted and sent to the suited plug-in. Javascript links are evaluated.
HTML_C	html	yes	no	no	yes	Fast C code generated in Perl. Not 100% accurate. Used for partial crawls.
Javascript	js	depends on the HTML plug-in	depends on the HTML plug-in	depends on the HTML plug-in	no	Javascripts are translated into the HTML code they are supposed to produce, and send to the HTML plug-in.
CSS	Style sheet	yes	not relevant	not relevant	no	
Flash	shockwave	yes	no	no	yes	Fast C code generated in Perl. Compressed flash documents are also handled. Only getUrl links are parsed.
DOC	MS Word documents	yes	yes	no	yes	May encounter false positives
RTF	Rich text format	yes	no	no	yes	May encounter false positives
Imagemap	Imagemap	yes	not relevant	not relevant	yes (line buffered)	
Raw	Plain text	yes	yes (look-behind window)	Yes (as is)	yes (line buffered)	Attempts are made to match URL in the text.
PLS	playlist	yes	yes (stream title)	not relevant	yes (line buffered)	streaming links
M3U	Winamp playlist	yes	yes (stream title)	not relevant	yes (line buffered)	streaming links

Real	Real Player playlist	yes	not relevant	not relevant	yes (line buffered)	streaming links
ASX	Media Player metafile	yes	yes (entry abstract and title)	not relevant	yes	streaming links

Implementing parsers for undocumented format is always difficult, but the web community often provides unofficial documentation or implementations on which to start our developments.
HTML is by far the most used plug-in. It is also the most difficult to handle correctly and in a fault tolerant yet rapid manner. It tries hard to mimic popular web browsers in their parsing and javascript interpretation (one of the trickiest things to handle).

1.4.4 Accurate Networking

Site crawlers are intended to be lightweight for both local and remote hardware (crawler and server). The site crawler uses a really straightforward networking approach: pages are downloaded in turn in a synchronous way[12]. To preserve server bandwidth, no parallel downloading is ever used, and further temporizations can be added to prevent the crawler from downloading more than a specified number of pages or amount of data in a given period of time. HTTP compression is implemented but rarely used because it involves a large computation overhead for the crawler, as well as for the server if contents are not pre-compressed. Nevertheless, most of the time HTTP compression is only applied for HTML pages. Other contents that are often self-compressed represent the majority of our needs in bandwidth (AV contents). Only one connection is kept open for each remote IP whose contents have to be downloaded from. If more than one IP exists for a given host, they are used sequentially each time a new connection is needed (i.e. each time a connection is closed), in a round-robin manner. Keeping connections open for a large number of requests is a major win for both the crawler and the remote server, opening TCP connection being quite expensive. HTTP/1.0 keep-alive and HTTP/1.1 chunk-encoding for dynamic contents are used whenever possible for this purpose. HTTP pipelining is not yet used, because most servers and proxies seem to fail handling it correctly, even when advertised as such. Connections are closed by the crawler if no request was made for a given time, or if it knows that no request will be made on this connection within a given time (we use 15 seconds). This strategy of keeping connections open only if they are needed and of forcing to reopen them more than would be necessary is made to adapt to most Web servers currently in use. Due to their multithreaded design, each open connection consumes memory and CPU resources of the server, whether the connection is active or not. Thereby, we have to constantly balance the cost of reopening a connection (for both the crawler and the remote server) with the cost we impose to the remote server by keeping unused connections open.

The balancing is made by grouping requests to fit with temporization parameters. Instead of applying temporization delays between each request, no delay is applied until the needed delay reaches a given duration. The connection is then closed during that period of time and reopened for the next group of requests. Connections closed by the server[13] are also considered as good points to apply the delay accumulated so far.

Nonetheless, connection handling and bandwidth management is not the only key to consider when designing a crawling strategy intended to put as little pressure as possible on the remote server. Nowadays web hosters spend most of their hardware resources generating dynamic contents[14]. To address this, dynamic contents are handled independently from other contents. At the parsing stage, dynamic-looking links are put in a different queue than static-looking ones. When subsequently traversing the queues, URLs are alternately taken from both queues in a way that ensures dynamic contents to be downloaded with the maximum possible static contents between them, leveling the load of the server.

[12] Anet is not used for this part of the site crawler, mostly for historical reasons.
[13] Most Apache servers we encountered were configured to close sockets after 100 requests.
[14] Despite modern dynamic web languages being embedded (or at least persistent) within the server and consuming much less resources than CGI scripts did, CPU and disk access requirements have moved toward database servers.

1.4.5 Crawling output

Crawling results are reported to the scheduler when the entire website has been crawled. Nevertheless, the scheduler can monitor each Site Crawler via HIP requests (uptime, memory consumption, etc.). Each Site Crawler also embeds a web server, listening on a different port, generating dynamic HTML monitoring pages.

During the download of a content, the crawler calculates its SHA256 signature, which identifies the content throughout the entire archiving process (cf. section 2.2.1). Each content signature is logged in a *signature file*. If the *signature file* of the former crawl was provided by the scheduler, the crawler only stores and analyzes new or updated contents.

Each document of the website is stored with a distinction between the signed content itself and its structure, that is to say the URL and the date at which it was downloaded. The structure is always stored, even if the content is already known and archived. The structure/content distinction is subsequently reproduced in the Replication storage layer (cf. section 2.2).

The site crawler also carries on analysis and indexing of new contents. Extracted texts (cf. section 1.4.3) are indexed in several ways:

- Local signatures are calculated inside a text, using chunk extraction algorithms [7]. These signatures are subsequently used to find similarities or updates in archived documents.
- Keywords are extracted using a variable length n-gram algorithm [8]. This keywords extraction is subsequently used for website clustering.
- The similarity of vocabulary between extracted texts and a set of *judge texts* is calculated using bloom filter techniques. This is used for language recognition.
- Special keywords or sentences given by the scheduler are looked up inside extracted texts. The occurrences of these keywords are used to determine if a website corresponds to a given topic.

External links, RSS [9], Atom [10] and AV stream URLs are also reported, together with data format analysis and other statistics, to the scheduler.

1.5 The scheduler

Each site crawler is an HIP slave for the scheduler. The scheduler maintains a pool of free/busy site crawlers on different machines. It can ask a site crawler to clone itself by forking a new one which in turn connects to the scheduler as a new available crawler. That way, only one site crawler needs to be launched on each machine, and can be duplicated afterward[15].

Basically, collecting a website unfolds in five steps:

1. The scheduler decides it is time to collect or recollect a particular website.
2. It chooses a free site crawler (or waits for one to finish its work) and asks it to collect the website, providing its delimitation (cf. section 1.4).
3. The site crawler is now registered as busy. No interaction is required between the scheduler and the site crawler until the crawl finishes. The scheduler can of course keep on collecting other websites in the meantime.
4. The site crawler informs the scheduler that the site has been crawled and locally archived along with reports information (external links, summaries, etc.).
5. Given this information, the scheduler applies some defined policies for further collects.

These tasks being very lightweight to handle, the scheduler will never become a bottleneck, even with thousands of agents running[16]. In fact, step 5 is handled by another type of HIP agents, also connected to the scheduler as slaves, called *connectivity agent*s, which apply the scheduler crawling policy by analyzing the results of the site crawlers.

We are currently using two scheduling policies:

- Extension scheduling: websites list constitution via focused crawling on a specific topic.
- Temporal scheduling: continuous archiving of a defined list of websites.

[15] Thanks to copy-on-write, forking is very efficient in terms of both speed and memory consumption on modern Unix-like operating systems.

[16] A HIP master benefits from scalable socket listener mechanisms provided by Anet to handle its slaves.

In practice, these two policies are handled in parallel during crawl campaigns. However, their issues and involvements will be discussed separately. For simplicity, we will also consider *connectivity agent*s as part of the scheduler for the next two sections.

1.5.1 Expansion scheduling

This scenario aims at finding all websites belonging to a defined web portion. The web portion is delimited by heuristic rules[17] and a list of websites used as starting points for the crawl campaign (seed). The start list is incrementally extended using focused crawling techniques with the website as the unit.

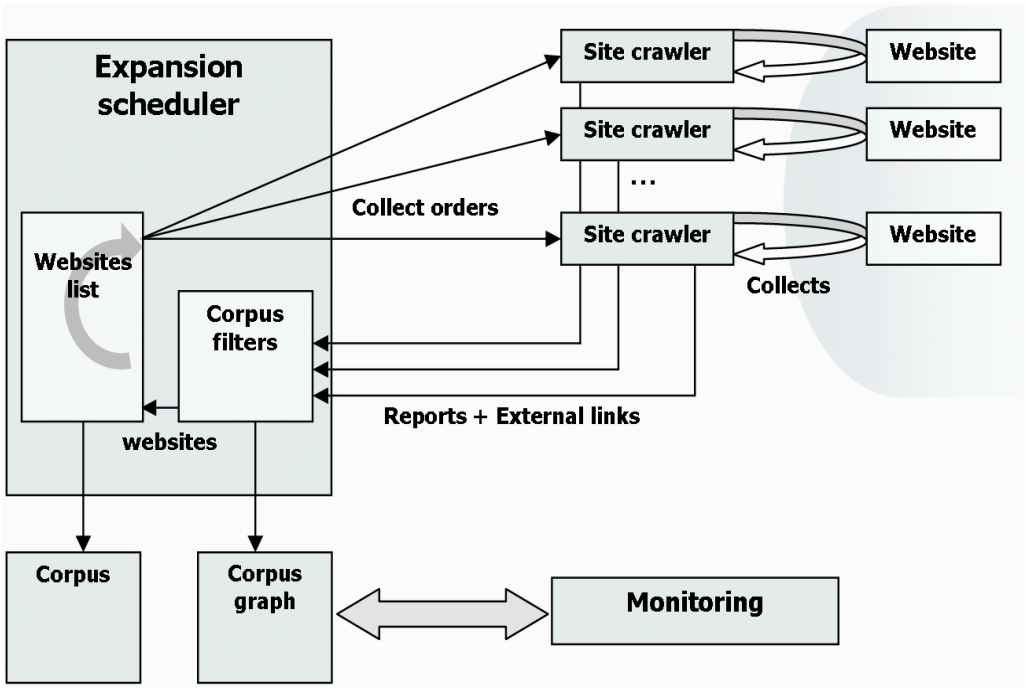

Figure 2: expansion scheduling process

Typical heuristic filters imply keywords, language recognition and technical format analysis.

Each time a website crawl is completed, the scheduler checks the reports of the crawler against its heuristic filters. If a website fails to pass the filters, it is put in a *blacklist* database table and subsequently ignored.

When a website passes the filters, it is given a relevancy rank and is considered as part of the web portion. Its external links are examined to see if they refer to already known websites. This is done by testing each link against the registered website delimitation *accept/reject* expressions in turn:

- If the link matches an *accept* expression and does not match the associated *reject* expression, the test is stopped[18] and the link is considered as referring to the website delimited by these expressions. If the website is not blacklisted, the *graph* database table is updated to reflect this link, together with the date.
- If the link dos not match any known website delimitation, it is considered as referring to a newly discovered website. The start URL and accept/reject expressions are guessed using heuristics taking into account hosting domain names for personal websites and other special URLs. A temporary website is registered and subsequently crawled.

The *graph* table is used to calculate hub/authority [11] ranks inside the web portion. The priority of each temporary website inside the scheduler queue is calculated using the relevancy and hub ranks of its referrers, as well as its authority rank.

[17] Heuristics are mostly based on linguistic and content type analysis.
[18] Delimitations of websites are disjunctives. Thereby, a given URL cannot belong to more than one website.

1.5.2 Temporal scheduling

In this scenario, the scheduler carries a continuous crawl of the web portion. Websites are crawled periodically and all data are archived. The crawl frequency of each website is automatically adjusted by analyzing the update rate between crawls, gradually making crawls in sync with actual updates.

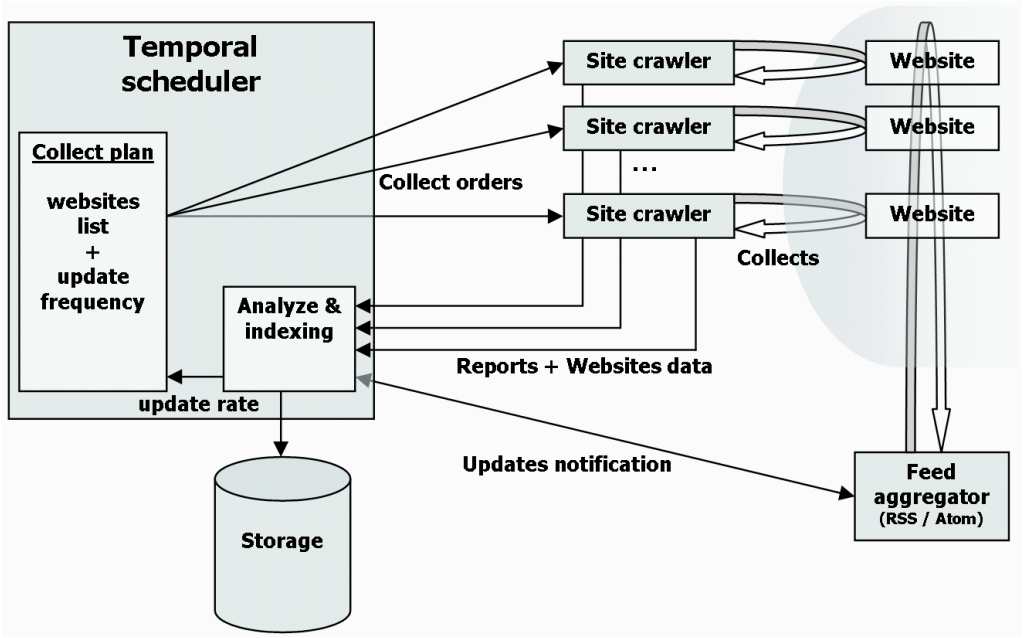

Figure 3: temporal scheduling process

When a website has to be crawled, the scheduler gives the crawler the *signature file* (cf. section 1.4.5) of its former crawl. Only new contents are stored by the site crawler. In case of quick update, the scheduler can also give the crawler the entire *structure file* of the former crawl, letting the site crawler go through the list of URLs with *if-modified-since* HTTP requests without the need to extract links from unaltered contents[19].

Updates between two crawls are analyzed using the rate of new data found in the website. Nevertheless, this is not done by simply taking into account new contents, because pages could have changed in a non pertinent way (random Ads, server-side date or temperature display, etc.). In fact, the update rate is calculated by analyzing the rate of new extracted text chunks (local signature extraction inside a text, cf. section 1.4.5) together with the total number of new contents. As a result, these two figures can specify if a website was updated by adding new pages or simply by appending new contents to existing ones. Moreover, update rates are calculated level by level, giving further indications on the needed maximum crawl depth.

The scheduler also uses a special type of HIP slave to track website changes: *Feed aggregators*. RSS and Atom links extracted by Sites Crawlers are reported as such to the Scheduler, enabling real-time change notification.

1.6 Streaming agent

In addition to Site Crawlers, Connectivity agents and Feed aggregators, other agents are used as slaves of the Scheduler. In this section, we describe the streaming agent responsible for the downloading of infinite streamed AV contents.

Inside a website, links to live streamed contents are extracted from playlists by Site Crawlers (cf. section 1.4.3) and reported as special streaming links.

[19] This type of crawling is not described in this article.

From the analysis of the relevance of the website inside the defined web portion, the scheduler can decide to collect new streamed contents.

Using Anet, streaming agents can handle a large number of concurrent stream downloads. Aside from format handling issues[20], downloading stream contents is a much simpler task than harvesting a website. It is merely a question of implementing proper protocols, a task in which Anet's stream API proved useful. Metadata embedded in the stream, such as song titles and descriptions, are logged and subsequently used for indexing purposes.

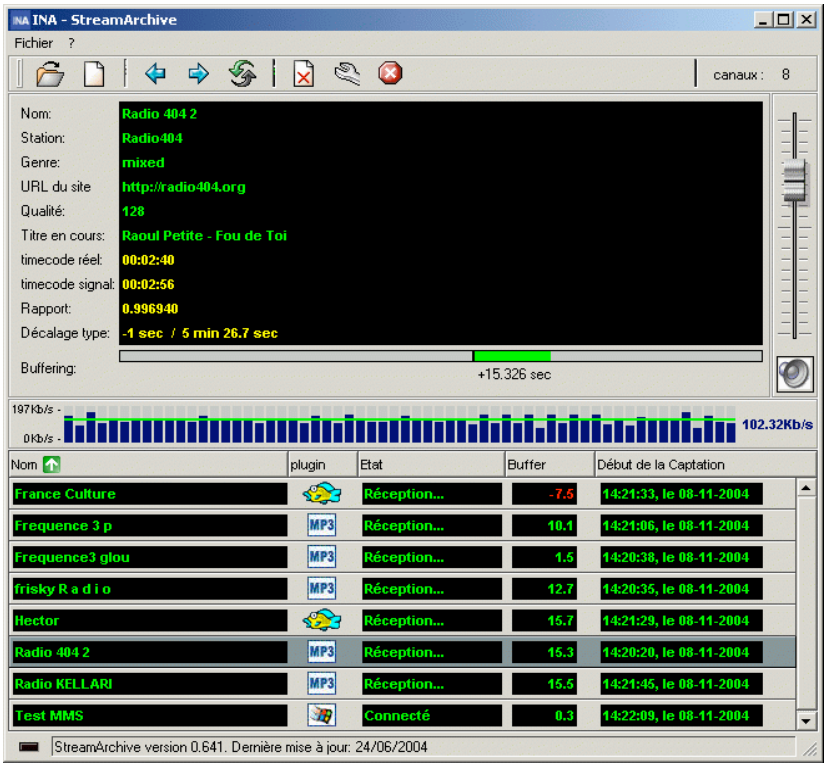

Figure 4: The interface of the streaming agent, downloading web radios

Nevertheless, downloading infinite streams uncovers some special issues that an archive system must take care of. Actually, live streamed contents are sent over the network at a bitrate defined by the encoding process. This rate is calculated using the clock of the encoding server as a basis, whose cadence can slightly differ from that of the receiver (clock skew). This results in an ever-increasing gap between collection duration and downloaded data size as time goes by, however small the clock difference is.

To address this problem, streaming agents regularly log the calculated time of each stream, based on the actual downloaded data, together with the real collection time. This information is used when accessing archived data (cf. section 2.2.2).

2 Storage model: the Replication Layer

The storage model is layer-based. Each layer presents a higher level of abstraction upon archive data, enriching web content with semantic, topological, and human analysis information. This section will focus on the lowest layer, called *replication layer* (RL), being the only layer to be fully scoped so far.

2.1 Requirements

RL aims at exactly reproducing content and structure of the targeted Web portion at any archived date. The plan is that it must be possible to reconstruct or restructure the whole storage model by only relying on this layer. This choice makes it the only layer that really needs to handle perennial issues in terms of software and hardware architecture. Hence, upper layers are free to concentrate on their own

[20] Format conversion and data encapsulation will not be dealt with in this article.

specificities using any suited technology, since we know we can rebuild any of them if it crashes or if its technology or architecture becomes obsolete.
Being the basis of the whole storage system, RL does not only need to be reliable, but also fast enough to handle new archived data at a sustained rate, as well as fast data retrieval for consultation or data mining purposes. Every addition, deletion, or evaluation of contents in archive will end up hitting this layer.
The third main requisite of RL is its scalability: it must be able to grow continuously to follow emergence of new contents on the targeted Web portion, shortly making Terabyte the basic storage unit.

These requirements paired with our empirical assumptions enforce several constraints in the design of the layer.

- In order to handle such storage capacities, the system needs to be distributed across clusters of machines. Single points of failure must be avoided to meet our speed and scalability requirements. Nevertheless, global system monitoring and administration must be considered as an essential part of the design.
- With the long term in view, the system will undoubtedly become heterogeneous in its hardware construction. The global system will be enriched and updated gradually as new storage requirements appear, or as existing machines fail. However, data must be accessed transparently through the whole system, and data safety must be monitored and analyzed constantly.
- Whenever possible, commercial off-the-shelf (COTS) hardware components will be preferred over specialized hardware components. Being more widely used, COTS products often have a higher value for money ratio, as well as a better tested and more robust behavior.
- As data durability is the main requisite, software architecture and data model must be simple, stable and open. Control needs to be kept on how our data are stored, we cannot thus depend on closed-source technologies, however established they could be. Simplicity will also always be favored over pure efficiency.

2.2 Basics

The storage hardware is made up of NAS nodes, typically commodity PCs running Linux, equipped with the maximum disks that can be handled. RAID systems are not necessary, redundancy being managed at the global system level (no RAID system is currently used, but the adding of reliability of such a system would be taken into account at the global level (cf. section 2.3.1). Similarly to the *Internet Archive* storage system [12], special care is taken on hardware choices as regards energy efficiency to minimize heat dissipation.

Data organization does not rely on a database engine. Everything merely consists in plain files stored flat in a normal file system. No big content concatenation files are used. The file system must thus be able to handle a large number of potentially small entries without wasting storage space or slowing down their access. ReiserFS [13] seems to be the most suited choice for this task. Version 3 is currently used, but switching to version 4 will be considered once integrated in Linux kernel.

2.2.1 Content files

Each unique archived content consists in a file named with its hexadecimal SHA256 signature, calculated during the crawl. The 256 bits signature is the guaranty for content uniqueness across the archive (ignoring storage redundancy), and collisions are statistically almost impossible, even far beyond quadrillions of contents in archive. In fact, SHA256 was not preferred to MD5 or SHA1 to avoid collisions (a 128 bits algorithm would have been good enough for our purpose), but to enable to build more consistent bloom filters [6]. by extracting more hash values from the signature. Intentional attacks must also be considered as a potential risk with short hash algorithms. Some are already broken or will certainly be in the future, such as MD5 [14]. Using a longer hash give more confidence on attacks avoidance for the lifetime of the archive.

Using one file per content releases some constraints about file consistency and increases possibilities of data recovery in case of disk failure, as well as easing data migrations.

2.2.2 Structure files

Sorted files are used to retrieve contents associated to given URL at given dates via binary search. Each archived URL involves a specific sorted file (again, the SHA256 of the URL is used as a name), every archived version consisting in a fixed size line inside the file with its date, the SHA256 of the content it refers to, and some Metadata (currently only used for AV contents, see below). As versions are

appended on every crawl, the file is naturally sorted by date[21]. Binary search enables exact, approximated, and range requests on dates efficiently without the need for a specific index.

Streamed AV contents are referred in structure files in a different manner: dates regularly logged during the download of the stream are used as different date entries for the resulting content file, with a time offset inside in the Metadata section. In fact, due to time shifting between real and calculated time with live streaming (cf. section 1.6), date entries (real logged time) and temporal offsets in Metadata (calculated time) can divaricate over time.

2.3 Cluster architecture

Nodes share their technical specificities with crawling nodes described in previous section, relying on Anet for networking and tasks dispatching and on HIP for inter-nodes RPC. Storage nodes are HIP slaves of a master node in charge of logging and marshalling the system. The master node is responsible for space allocation across storage nodes, data redundancy, and recovery in case of node failure. Each slave node logs the files which are being added to it along with other monitoring information on the HIP stream, so that the master knows which node stores which content, how safe it is, and how much space is left. The master node also accepts another kind of slave, called index node, in charge of maintaining indexes to files and allocable space among nodes. Index nodes are almost only used for data insertion or data restoration in case of failure of the master node. HTTP and multicasting is used for data access. This model is similar to the Google file system (GFS [15]), except that data access do not have to systematically go through the master. This was an important requirement given the number of small files that need to be accessed, whereas GFS uses big compound files.

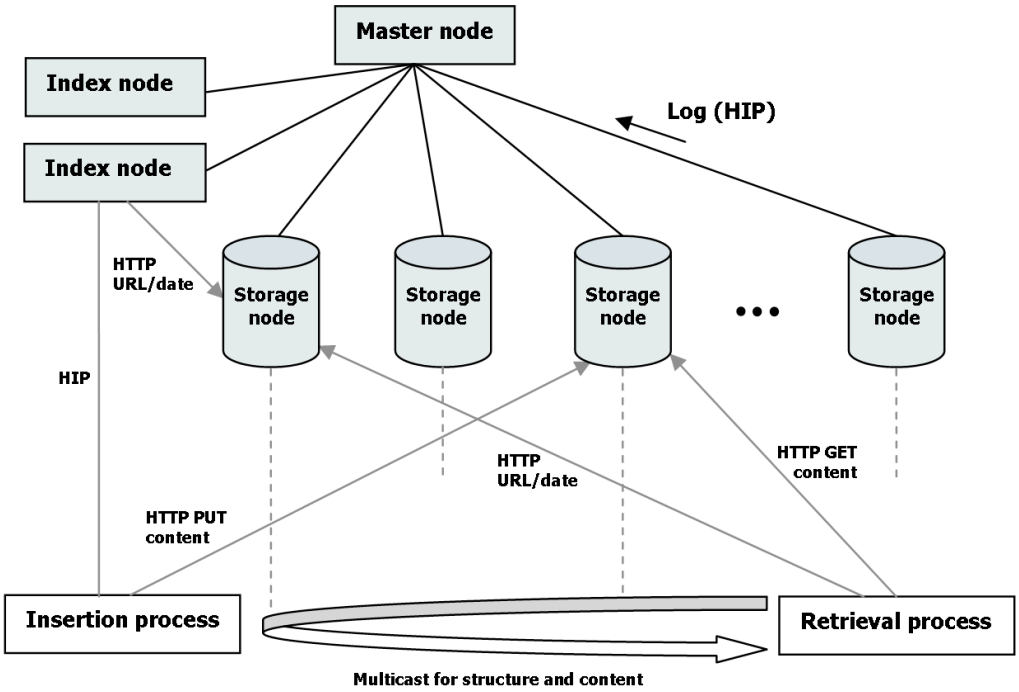

Figure 5: Storage cluster and access scenarios

2.3.1 Data insertion

Data insertion is handled through Index nodes. The crawling process in charge of inserting new data[22] opens an HIP connection to the master node, which redirects it to an index node (HIP slave redirection). The HIP connection is then kept open to the index node. For each insertion, given the URL

[21] Dates of former lines are always checked on insertion. Some file manipulation might be needed in case of belated data insertion, but this is rarely the case and only involves small portions of the file to be moved.

[22] This task is currently handled by the connectivity agent

of the document, the SHA256 of the content it refers to, and the date of its crawl, the index node directly contacts the storage node that holds the structure file associated with the URL to append a new entry. Using the first 128 bits of the SHA256 as a key, the index node then queries a local database to determine if the content file already exists in the archive. If the content is new, the index node indicates the crawling process on which node server the content has to be put, based on storage space availability and group subscriptions (cf. section 2.3.2, on the use of multicast groups for data retrieval). The content file is put on the designed storage node by connecting to its Anet HTTP server.

It is clear that the number of index nodes must be increased in concordance with the intended crawling speed.

Each storage node maintains a bloom filter of the content and structure files it stores. Although it is impossible to remove items from a bloom filter, content deletion (mainly due to files reallocation across nodes) does not imply the filter to be rebuilt: an in-memory hash is used to denote files that are no longer stored on the node but still pass the filter. Testing for a file means testing the bloom filter and the deletion hash in turn. A fresh filter is rebuilt when the number of elements in the deletion hash exceeds a given limit.

New inserted contents are marked as new by the index node, and are subsequently duplicated with a given replica count to other nodes. This principle is similar to the one used in MogileFS [16] and is meant to overcome conventional RAID issues. The replica count is currently roughly fixed for all contents (small contents are slightly favored though). Nonetheless, further investigation reveal four points to be considered for this calculation in the near future. Namely:

- The content itself. For technical reasons, we shall try to automatically estimate an informational/storage ratio for each content. A small PDF document will thus gain more replicas than a big high resolution movie.
- The representativeness of the content. Contents that are referred to from many different URL or many dates shall be favored.
- The hardware it is stored on. For example, data stored on a RAID5 system shall be considered safer.
- The location of the storage. Files duplicated on geographically distant machines and/or maintained by different persons shall be considered safer.

Structure files are also replicated as backups (only one structure file is active) with a high replica count.

In case of failure of a storage node, the master node is immediately informed in consequence of the corresponding HIP connection being closed. Logs from the missing node are then analyzed and replica counts of the files it stored are reevaluated.

2.3.2 Data retrieval

Data retrieval is entirely decentralized. Contents are discovered by multicasting a request to a specific group IP calculated by setting the last IP byte to the first SHA256 byte of the requested content on a defined IP range (230.0.0.0 to 230.0.0.255 for example). Similarly, storage nodes join each of the 256 groups that matches any content they store[23]. Benefits from this technique become evident when contents reparation among storage nodes is well controlled, reducing request to each node up to a 256^{th} compared to a global broadcasting architecture. With a small number of storage nodes, adjunction of new nodes may require contents to be rebalanced and IP subscriptions rehashed for efficiency. Nevertheless, this content repartition is only a matter of optimization and is not mandatory: nodes could also subscribe to all of the 256 possible groups and receive every request. Request handling is fairly light for the node: a small bloom filter is checked, a negative test definitely meaning that the content is not stored on the node. Due to false-positive risks, a positive test leads the node to check for the existence of the file in the file system. Nodes that hold the requested file respond to the request originator, signaling that the file can be got from their HTTP server.

The same technique is used for structure files, except that a binary search is also done on the file, increasing CPU and disk operations on the node.

[23] Up to 256 subscriptions can be made, using several sockets if OS or hardware limits prevent us to reach that number with a single socket. Typical limit seems to be arbitrary set to 20 in most OS.

Suppose a consultation server wants to fetch the content of an URL at a given date in the archive, the complete retrieval takes place in seven steps:

1. The consultation server multicasts a structure file request to the group corresponding to the SHA256 of the URL.
2. Storage nodes that subscribed to the multicast group receive the request and look up the file. The Storage node holding the active structure file responds with an HTTP address.
3. The consultation server requests the given address with the date it is looking for in GET parameter (Several types of requests are possible).
4. The storage node proceeds to a binary search to find the temporally closest version. Once found, the SHA256 of the content file is returned to the consultation server, along with the exact date that was found.
5. If the date meets the needs of the consultation server, a second multicast request is issued to locate the content file.
6. Storage nodes holding the content file return their HTTP address (due to redundancy, several nodes can hold the file).
7. The consultation server chooses a storage node and fetches the content file by issuing a GET request to its HTTP server.[24].

For both structure and content file requests, if no response is received within a short period of time the consultation server considers the file as missing. In terms of HTTP vocabulary, it should conclude to a 404 error in case of structure file timeout (the URL is unknown), or a 500 error for content file timeout (the URL exists and refers to a content that is missing; this is an internal archive error).

Scalability of this system allows envisaging massive data mining scenarios, with multiple consultation servers accessing the archive concurrently.

2.4 Reconstructing an archived Web

Underlying technical principles of data access being settled, browsing scenarios of the archive will be dealt with in this section.

2.4.1 Link modification

As contents are stored unmodified, the links they might include keep refering to real web contents. To enable the browsing of these contents, we need to redirect these links to an archived version instead of the current Web one. The usual solution for this problem consists in modifying the links themselves, making them link back to the archive. Most website copy programs operate this modification during the crawl in a destructive manner, in order to enable the copy to be browsed locally on hard-drive or CD. *Internet Archive* also uses such an approach for its web interface [17], modifying links on the fly before sending the content to the client browser. Besides being quite CPU intensive, link modification is also a particularly perilous task. While some links are already quite difficult to interpret and follow during the crawling process (javascript, flash, applets, etc.), their modification appears much harder yet. As a consequence, some links might still refer to online contents, leading to possible inconsistencies if they disappear or get modified. This solution was thus considered improper for a legal deposit purpose.

2.4.2 Proxy Browsing

In the approach we investigated, the problem is considered from the opposite. Instead of making web links refer to the archive, the archive is made to act as if it was the web. Practically, this is done by implementing a consultation server into an HTTP proxy. Browsers that need to access the archive get configured to use the consultation server as their web proxy. Thereby, every request sent to the web by the browser is intercepted by the consultation server, and an archived version of the requested resource at a given date is sent back without the need to modify any link. The browser interacts with the archive exactly as it would have with the web at that given date, automatically parsing links to request embedded elements. Responses sent by the consultation server correspond to those received by site crawlers when they issued requests corresponding to those sent by the browser, at the closest date. If no response can be

[24] Being an Anet component, the HTTP server relies on *sendfile* for file transmission on systems that support it. Thereby, sending files is a very lightweight task.

found for a given request, it is interpreted by the consultation server as a fault during the crawling process (missed link). In such situations, the special *fault content* is sent to the browser, putting emphasis on any potential inconsistency in the rendering of a page with an explicit message[25].

2.4.3 Archive harbors

The date at which the archive is browsed as well as other parameters are configured via special virtual websites called *archive harbors*. These websites do not exist on the web or within the archive at any date, but consist in purely virtual sites directly interfacing with the consultation server. To avoid collisions with real websites, *Archive harbors* are regrouped under the made up *dlweb* (web legal deposit) top level domain name. Parameters configured via these harbors are registered in the consultation server independently for each browser IP address. The user can thus browse the web at a given date, change this date in an *archive harbor*, and resume his browsing with the new date[26]. This can be seen as a corridor of doors opening on different archived versions of the web inside the URL/date matrix.

Higher storage layers also have specific *archive harbors* to interact with their data, including search-engines[27], directories, maps, etc.

2.4.4 Data mining

The proxy interface makes it possible to use automated HTTP tools for mining data. In fact, any crawler-based system made for the web can be used on the archive this way, interacting with the archive as they would have with the web. This is really what RL is made for: backup web contents and render them as they were at a given date. Based on this, it would even be possible to utterly re-crawl and restore the archive with a completely different storage model. This is a capital point for the durability of the archive.

Automated crawling tools can also be used to enrich higher storage layers or check archive sanity. Moreover, a researcher can carry its own in-depth studies on specific topics using web analysis tools such as random walkers or graph robots.

The implementation of a SOAP interface for archive parameterization is foreseen, avoiding the need for screen scraping *archive harbors* websites when programming specialized crawlers that need to interact with storage layers (date manipulation, full-text search, backlinks, similar contents, etc.). This SOAP API will also provide tools to extract information from documents (links extraction, text extraction, etc.), easing rapid development of archive crawlers in any programming language.

Perspectives

Albeit conceived for a specific purpose, the system we described can also be used for other Web archiving applications. In fact, we are already carrying on experimentations our archiving system on other web topics such as migrating usages on the Web [18]. Moreover, the crawling system described in this article is now also used by the RTGI team of the Compiègne Technological University to carry on their experimentations [19].

With the French Web legal deposit being legislated, a period of approximately three years will be necessary to gradually set up a fully functional archiving process. Important parts of the project, such as upper storage layers or visualizing tools, need further developments and experiments. Existing solutions also need to be improved. Admittedly, building an archiving model for the Web is a never-ending task, striving against technical as well as conceptual evolutions.

Acknowledgments

I would like to thank Jérôme Thièvre for the development of specialized graph visualization tools for the focused scheduler, Hugo Zanghi for his work on the Scheduler implementation, Jean-Philippe Poli and Marianne Drugeon for having reviewed this paper, Bruno Bachimont, Jean-Michel Rodes, Geneviève Piejut, Nicolas Delaforge and Younes Hafri for their contribution to the project.

[25] The format of the content is guessed upon the requested URL. *Fault content* currently exists as image, HTML, plain text, Javascript (by way of document.write), CSS, Flash, PDF, and MS Word document.

[26] The user can also choose to browse the current web, making the consultation server act as a real web proxy server.

[27] This can be seen as a substitute for web search-engines that are not archived.

References

1 Marc Najork, Allan Heydon, "High-PerformanceWeb Crawling Mercator", 2001

2 Vladislav Shkapenyuk, Torsten Suel, "Design and implementation of a High-Performance Distributed Web Crawler", 2001

3 Gordon Mohr, Michele Kimpton, Micheal Stack, Igor Ranitovic, "Introduction to Heritrix, an archival quality web crawler", 2004

4 POE, http://poe.perl.org/ (visited on 1 June 2005)

5 YAML, http://yaml.org/ (visited on 1 June 2005)

6 Burton H. Bloom, "Space/time trade-offs in hash coding with allowable errors", 1970

7 Nevin Heintze, "Scalable Document Fingerprinting", 1996

8 William B. Cavnar, John M. Trenkle, "N-Gram-Based Text Categorization", 1994

9 RSS, http://blogs.law.harvard.edu/tech/rss (visited on 1 June 2005)

10 Atom, http://www.atomenabled.org/ (visited on 1 June 2005)

11 J. M. Kleinberg, "Authoritative Sources in a Hyperlinked Environment", 1999

12 Petabox, http://www.archive.org/web/petabox.php (visited on 1 June 2005)

13 ReiserFS, http://namesys.com/ (visited on 1 June 2005)

14 Dan Kaminsky, "MD5 To Be Considered Harmful Someday", 2004

15 Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, "The Google File System", 2003

16 MogileFS, Brad Fitzpatrick, Danga, URL: http://danga.com/mogilefs/ (visited on 1 June 2005)

17 Internet Archive, http://archive.org/ (visited on 1 June 2005)

18 Julien LAFLAQUIERE, Sylvie GANGLOFF, Claire SCOPSI, Thomas GUIGNARD, Ralitza SOULTANOVA, Monika SALZBRUNN, Virginie BEAUJOUAN et Dana DIMINESCU, "Archiver le Web sur les migrations : quelles approches techniques et scientifiques", Migrance 2005

19 RTGI, "Le web et le débat sur la constitution européenne en France", http://www.utc.fr/rtgi/index.php?rubrique=1&sousrubrique=0&study=constitution (visited on 1 June 2005)