High Performance Crawling System

Younès Hafri
Ecole Polytechnique de Nantes
Institut National de l'Audiovisuel
4, avenue de l'Europe
94366 Bry sur Marne - cedex, France
yhafri@ina.fr

Chabane Djeraba
Laboratoire d'Informatique Fondamentale Lille
UMR CNRS 8022- Batiment M3
59655 Villeneuve d'Ascq Cédex - France
djeraba@lifl.fr

ABSTRACT

In the present paper, we will describe the design and implementation of a real-time distributed system of Web crawling running on a cluster of machines. The system crawls several thousands of pages every second, includes a high-performance fault manager, is platform independent and is able to adapt transparently to a wide range of configurations without incurring additional hardware expenditure. We will then provide details of the system architecture and describe the technical choices for very high performance crawling. Finally, we will discuss the experimental results obtained, comparing them with other documented systems.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Distributed programming; C.4 [**Performance of Systems**]: Fault tolerance; H.3.7 [**Digital Libraries**]: Systems issues

Keywords

Web Crawler, Hierarchical Cooperation, High Availability System

1. INTRODUCTION

With the World Wide Web containing the vast amount of information (several thousands in 1993, 3 billion today) that it does and the fact that it is ever expanding, we need a way to find the right information (multimedia of textual). We need a way to access the information on specific subjects that we require. To solve the problems above several programs and algorithms were designed that index the web, these various designs are known as search engines, spiders, crawlers, worms or knowledge robots graph in its simplest terms. The pages are the nodes on the graph and the links are the arcs on the graph. What makes this so difficult is the vast amount of data that we have to handle, and then we must also take into account the fact that the World Wide Web is constantly growing and the fact that people are constantly updating the content of their web pages.

Any High performance crawling system should offer at least the following two features. Firstly, it needs to be equipped with an intelligent navigation strategy, i.e. enabling it to make decisions regarding the choice of subsequent actions to be taken (pages to be downloaded etc). Secondly, its supporting hardware and software architecture should be optimized to crawl large quantities of documents per unit of time (generally per second). To this we may add fault tolerance (machine crash, network failure etc.) and considerations of Web server resources.

Recently we have seen a small interest in these two field. Studies on the first point include crawling strategies for important pages [9, 17], topic-specific document downloading [5, 6, 18, 10], page recrawling to optimize overall refresh frequency of a Web archive [8, 7] or scheduling the downloading activity according to time [22]. However, little research has been devoted to the second point, being very difficult to implement [20, 13]. We will focus on this latter point in the rest of this paper.

Indeed, only a few crawlers are equipped with an optimized scalable crawling system, yet details of their internal workings often remain obscure (the majority being proprietary solutions). The only system to have been given a fairly in-depth description in existing literature is *Mercator* by Heydon and Najork of DEC/Compaq [13] used in the AltaVista search engine (some details also exist on the first version of the Google [3] and Internet Archive [4] robots).

Most recent studies on crawling strategy fail to deal with these features, contenting themselves with the solution of minor issues such as the calculation of the number of pages to be downloaded in order to maximize/minimize some functional objective. This may be acceptable in the case of small applications, but for *real time*[1] applications the system must deal with a much larger number of constraints. We should also point out that little academic research is concerned with high performance search engines, as compared with their commercial counterparts (with the exception of the WebBase project [14] at Stanford).

In the present paper, we will describe a very high availability, optimized and distributed crawling system. We will use the system on what is known as *breadth-first* crawling, though this may be easily adapted to other navigation strategies. We will first focus on input/output, on management of network traffic and robustness when changing scale. We will also discuss download policies in

[1]"Soft" real time

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MIR'04, October 15–16, 2004, New York, New York, USA.
Copyright 2004 ACM 1-58113-940-3/04/0010...$5.00.

terms of speed regulation, fault management by supervisors and the introduction/suppression of machine nodes without system restart during a crawl.

Our system was designed within the experimental framework of the *Dépôt Légal du Web Français (French Web Legal Deposit)*. This consists of archiving only multimedia documents in French available on line, indexing them and providing ways for these archives to be consulted. Legal deposit requires a real crawling strategy in order to ensure site continuity over time. The notion of registration is closely linked to that of archiving, which requires a suitable strategy to be useful. In the course of our discussion, we will therefore analyze the implication and impact of this experimentation for system construction.

2. STATE OF THE ART

2.1 Prerequisites of a Crawling System

In order to set our work in this field in context, listed below are definitions of services that should be considered the minimum requirements for any large-scale crawling system.

- Flexibility: as mentioned above, with some minor adjustments our system should be suitable for various scenarios. However, it is important to remember that crawling is established within a specific framework: namely, Web legal deposit.
- High Performance: the system needs to be scalable with a minimum of one thousand pages/second and extending up to millions of pages for each run on low cost hardware. Note that here, the quality and efficiency of disk access are crucial to maintaining high performance.
- Fault Tolerance: this may cover various aspects. As the system interacts with several servers at once, specific problems emerge. First, it should at least be able to process invalid HTML code, deal with unexpected Web server behavior, and select good communication protocols etc. The goal here is to avoid this type of problem and, by force of circumstance, to be able to ignore such problems completely. Second, crawling processes may take days or weeks, and it is imperative that the system can handle failure, stopped processes or interruptions in network services, keeping data loss to a minimum. Finally, the system should be *persistent*, which means periodically switching large data structures from memory to the disk (e.g. restart after failure).
- Maintainability and Configurability: an appropriate interface is necessary for monitoring the crawling process, including download speed, statistics on the pages and amounts of data stored. In online mode, the administrator may adjust the speed of a given crawler, add or delete processes, stop the system, add or delete system nodes and supply the black list of domains not to be visited, etc.

2.2 General Crawling Strategies

There are many highly accomplished techniques in terms of Web crawling strategy. We will describe the most relevant of these here.

- Breadth-first Crawling: in order to build a wide Web archive like that of the Internet Archive [15], a crawl is carried out from a set of Web pages (initial URLs or seeds). A breadth-first exploration is launched by following hypertext links leading to those pages directly connected with this initial set. In fact, Web sites are not really browsed breadth-first and various restrictions may apply, e.g. limiting crawling processes to within a site, or downloading the pages deemed most interesting first[2]
- Repetitive Crawling: once pages have been crawled, some systems require the process to be repeated periodically so that indexes are kept updated. In the most basic case, this may be achieved by launching a second crawl in parallel. A variety of heuristics exist to overcome this problem: for example, by frequently relaunching the crawling process of pages, sites or domains considered important to the detriment of others. A good crawling strategy is crucial for maintaining a constantly updated index list. Recent studies by Cho and Garcia-Molina [8, 7] have focused on optimizing the update frequency of crawls by using the history of changes recorded on each site.
- Targeted Crawling: more specialized search engines use crawling process heuristics in order to target a certain type of page, e.g. pages on a specific topic or in a particular language, images, mp3 files or scientific papers. In addition to these heuristics, more generic approaches have been suggested. They are based on the analysis of the structures of hypertext links [6, 5] and techniques of learning [9, 18]: the objective here being to retrieve the greatest number of pages relating to a particular subject by using the minimum bandwidth. Most of the studies cited in this category do not use high performance crawlers, yet succeed in producing acceptable results.
- Random Walks and Sampling: some studies have focused on the effect of random walks on Web graphs or modified versions of these graphs via sampling in order to estimate the size of documents on line [1, 12, 11].
- Deep Web Crawling: a lot of data accessible via the Web are currently contained in databases and may only be downloaded through the medium of appropriate requests or forms. Recently, this often-neglected but fascinating problem has been the focus of new interest. The *Deep Web* is the name given to the Web containing this category of data [9].

  Lastly, we should point out the acknowledged differences that exist between these scenarios. For example, a breadth-first search needs to keep track of all pages already crawled. An analysis of links should use structures of additional data to represent the graph of the sites in question, and a system of classifiers in order to assess the pages' relevancy [6, 5]. However, some tasks are common to all scenarios, such as

[2]See [9] for the heuristics that tend to find the most important pages first and [17] for experimental results proving that breadth-first crawling allows the swift retrieval of pages with a high PageRank.

respecting robot exclusion files (robots.txt), crawling speed, resolution of domain names ...

In the early 1990s, several companies claimed that their search engines were able to provide complete Web coverage. It is now clear that only partial coverage is possible at present. Lawrence and Giles [16] carried out two experiments in order to measure coverage performance of data established by crawlers and of their updates. They adopted an approach known as *overlap analysis* to estimate the size of the Web that may be indexed (See also Bharat and Broder 1998 on the same subject). Let W be the total set of Web pages and $W_a \subset W$ and $W_b \subset W$ the pages downloaded by two different crawlers a and b. What is the size of W_a and W_b as compared with W? Let us assume that uniform samples of Web pages may be taken and their membership of both sets tested. Let $P(W_a)$ and $P(W_b)$ be the probability that a page is downloaded by a or b respectively. We know that:

$$P(W_a \cap W_b | W_b) = \frac{W_a \cap W_b}{|W_b|} \tag{1}$$

Now, if these two crawling processes are assumed to be independent, the left side of equation 1 may be reduced to $P(W_a)$, that is data coverage by crawler a. This may be easily obtained by the intersection size of the two crawling processes. However, an exact calculation of this quantity is only possible if we do not really know the documents crawled. Lawrence and Giles used a set of controlled data of 575 requests to provide page samples and count the number of times that the two crawlers retrieved the same pages. By taking the hypothesis that the result $P(W_a)$ is correct, we may estimate the size of the Web as $|W_a|/P(W_a)$. This approach has shown that the Web contained at least 320 million pages in 1997 and that only 60% was covered by the six major search engines of that time. It is also interesting to note that a single search engine would have covered only 1/3 of the Web. As this approach is based on observation, it may reflect a visible Web estimation, excluding for instance pages behind forms, databases etc. More recent experiments assert that the Web contains several billion pages.

2.2.1 Selective Crawling

As demonstrated above, a single crawler cannot archive the whole Web. The fact is that the time required to carry out the complete crawling process is very long, and impossible given the technology currently available. Furthermore, crawling and indexing very large amounts of data implies great problems of scalability, and consequently entails not inconsiderable costs of hardware and maintenance. For maximum optimization, a crawling system should be able to recognize relevant sites and pages, and restrict itself to downloading within a limited time.

A document or Web page's relevancy may be officially recognized in various ways. The idea of selective crawling may be introduced intuitively by associating each URL u with a score calculation function $s_\theta^{(\xi)}$ respecting relevancy criterion ξ and parameters θ. In the most basic case, we may assume a Boolean relevancy function, i.e. $s(u) = 1$ if the document designated by u is relevant and $s(u) = 0$ if not. More generally, we may think of $s(d)$ as a function with real values, such as a conditional probability that a document belongs to a certain category according to its content. In all cases, we should point out that the score calculation function depends only on the URL and ξ and not on the time or state of the crawler.

A general approach for the construction of a selective crawler consists of changing the URL insertion and extraction policy in the queue Q of the crawler. Let us assume that the URLs are sorted in the order corresponding to the value retrieved by $s(u)$. In this case, we obtain the *best-first* strategy (see [19]) which consists of downloading URLs with the best scores first). If $s(u)$ provides a good relevancy model, we may hope that the search process will be guided towards the best areas of the Web.

Various studies have been carried out in this direction: for example, limiting the search depth in a site by specifying that pages are no longer relevant after a certain depth. This amounts to the following equation:

$$s_\theta^{(depth)}(u) = \begin{cases} 1, & \text{if } |root(u) \approx u| < \delta \\ 0, & \text{else} \end{cases} \tag{2}$$

where $root(u)$ is the root of the site containing u. The interest of this approach lies in the fact that maximizing the search breadth may make it easier for the end-user to retrieve the information. Nevertheless, pages that are too deep may be accessed by the user, even if the robot fails to take them into account.

A second possibility is the estimation of a page's popularity. One method of calculating a document's relevancy would relate to the number of backlinks.

$$s_\theta^{(backlinks)}(u) = \begin{cases} 1, & \text{if } indegree(u) > \tau \\ 0, & \text{else} \end{cases} \tag{3}$$

where τ is a threshold.

It is clear that $s_\theta^{(backlinks)}(u)$ may only be calculated if we have a complete site graph (site already downloaded beforehand). In practice, we make take an approximate value and update it incrementally during the crawling process. A derivative of this technique is used in Google's famous *PageRank* calculation.

3. OUR APPROACH: THE DOMINOS SYSTEM

As mentioned above, we have divided the system into two parts: *workers* and *supervisors*. All of these processes may be run on various operating systems (Windows, MacOS X, Linux, FreeBSD) and may be replicated if need be. The workers are responsible for processing the URL flow coming from their supervisors and for executing crawling process tasks in the strict sense. They also handle the resolution of domain names by means of their integrated DNS resolver, and adjust download speed in accordance with node policy. A worker is a light process in the Erlang sense, acting as a fault tolerant and highly available *HTTP* client. The process-handling mode in Erlang makes it possible to create several thousands of workers in parallel.

In our system, communication takes place mainly by sending asynchronous messages as described in the specifications for Erlang language. The type of message varies according to need: character string for short messages and binary format for long messages (large data structures or files). Disk access is reduced to a minimum as far as possible and structures are stored in the real-time Mnesia [3] database that forms

[3] *http://www.erlang.org/doc/r9c/lib/mnesia-4.1.4/doc/html/*

a standard part of the Erlang development kit. Mnesia's features give it a high level of homogeneity during the base's access, replication and deployment. It is supported by two table management modules *ETS* and *DETS*. ETS allows tables of values to be managed by random access memory, while DETS provides a persistent form of management on the disk. Mnesia's distribution faculty provides an efficient access solution for distributed data. When a worker moves from one node to another (code migration), it no longer need be concerned with the location of the base or data. It simply has to read and write the information transparently.

```
 1 loop(InternalState) -> % Supervisor main
 2                         % loop
 3 receive {From,{migrate,Worker,Src,Dest}} ->
 4  % Migrate the Worker process from
 5  % Src node to Dest node
 6  spawn(supervisor,migrate,
 7         [Worker,Src,Dest]),
 8  % Infinite loop
 9   loop(InternalState);
10
11   {From,{replace,OldPid,NewPid,State}} ->
12    % Add the new worker to
13    % the supervisor state storage
14    NewInternalState =
15     replace(OldPid,NewPid,InternalState),
16    % Infinite loop
17    loop(NewInternalState);
18      ...
19  end.
20
21  migrate(Pid,Src,Dest) -> % Migration
22                            % process
23  receive
24    Pid ! {self(), stop},
25    receive
26     {Pid,{stopped,LastState}} ->
27        NewPid = spawn{Dest,worker,proc,
28                      [LastState]},
29        self() ! {self(), {replace,Pid,
30                  NewPid,LastState}};
31    {Pid,Error} -> ...
32  end.
```

Listing 1: Process Migration

Code 1 describes the migration of a worker process from one node *Src* to another *Dest*.[4] The supervisor receives the migration order for process *Pid* (line 4). The migration action is not blocking and is performed in a different Erlang process (line 7). The supervisor stops the worker with the identifier *Pid* (line 25) and awaits the operation result (line 26). It then creates a remote worker in the node *Dest* with the latest state of the stopped worker (line 28) and updates its internal state (lines 30 and 12).

3.1 Dominos Process

The Dominos system is different from all the other crawling systems cited above. Like these, the Dominos offering is on distributed architecture, but with the difference of being totally *dynamic*. The system's dynamic nature allows its architecture to be changed as required. If, for instance, one of the cluster's nodes requires particular maintenance, all of the processes on it will *migrate* from this node to another. When servicing is over, the processes revert automatically to their original node. Crawl processes may change pool so as to reinforce one another if necessary. The addition or deletion of a node in the cluster is completely transparent in its execution. Indeed, each new node is created containing a completely blank system. The first action to be undertaken is to search for the generic server in order to obtain the parameters of the part of the system that it is to belong to. These parameters correspond to a *limited view* of the whole system. This enables Dominos to be deployed more easily, the number of messages exchanged between processes to be reduced and allows better management of exceptions. Once the generic server has been identified, binaries are sent to it and its identity is communicated to the other nodes concerned.

[4]The character % indicates the beginning of a comment in Erlang.

- Dominos Generic Server (GenServer): Erlang process responsible for managing the process identifiers on the whole cluster. To ensure easy deployment of Dominos, it was essential to mask the denominations of the process identifiers. Otherwise, a minor change in the names of machines or their IP would have required complete reorganization of the system. GenServer stores globally the identifiers of all processes existing at a given time.

- Dominos RPC Concurrent (cRPC): as its name suggests, this process is responsible for delegating the execution of certain remote functions to other processes. Unlike conventional RPCs where it is necessary to know the node and the object providing these functions (services), our RPCC completely masks the information. One need only call the function, with no concern for where it is located in the cluster or for the name of the process offering this function. Moreover, each RPCC process is concurrent, and therefore manages all its service requests in parallel. The results of remote functions are governed by two modes: blocking or non-blocking. The calling process may therefore await the reply of the remote function or continue its execution. In the latter case, the reply is sent to its mailbox. For example, no worker knows the process identifier of its own supervisor. In order to identify it, a worker sends a message to the process called *supervisor*. The RPCC deals with the message and searches the whole cluster for a *supervisor* process identifier, starting with the local node. The address is therefore resolved without additional network overhead, except where the supervisor does not exist locally.

- Dominos Distributed Database (DDB): Erlang process responsible for Mnesia real-time database management. It handles the updating of crawled information, crawling progress and the assignment of URLs to be downloaded to workers. It is also responsible for replicating the base onto the nodes concerned and for the persistency of data on disk.

- Dominos Nodes: a node is the physical representation of a machine connected (or disconnected as the case may be) to the cluster. This connection is considered in the most basic sense of the term, namely a simple plugging-in (or unplugging) of the network outlet. Each node clearly reflects the dynamic character of the Dominos system.

- Dominos Group Manager: Erlang process responsible for controlling the smooth running of its child processes (supervisor and workers).
- Dominos Master-Supervisor Processes: each group manager has a single master process dealing with the management of crawling states of progress. It therefore controls all the slave processes (workers) contained within it.
- Dominos Slave-Worker Processes: workers are the lowest-level elements in the crawling process. This is the very heart of the Web client wrapping the libCURL.

With Dominos architecture being completely dynamic and distributed, we may however note the hierarchical character of processes within a Dominos node. This is the only way to ensure very high fault tolerance. A group manager that fails is regenerated by the node on which it depends. A master process (supervisor) that fails is regenerated by its group manager. Finally, a worker is regenerated by its supervisor. As for the node itself, it is controlled by the Dominos kernel (generally on another remote machine). The following code describes the regeneration of a worker process in case of failure.

```
1  % Activate error handling
2  process_flag(trap_exit,true),
3   ...
4  loop(InternalState) -> % Supervisor main loop
5  receive
6   {From,{job,Name,finish}, State} ->
7    % Informe the GenServer that the download is ok
8    ?ServerGen ! {job,Name,finish},
9
10   % Save the new worker state
11   NewInternalState=save_state(From,State,InternalState),
12
13   % Infinite loop
14   loop(NewInternalState);
15   ...
16   {From,Error} -> % Worker crash
17   % Get the last operational state before the crash
18   WorkerState = last_state(From,InternalState),
19
20   % Free all allocated resources
21   free_resources(From,InternalState),
22
23   % Create a new worker with the last operational
24   % state of the crashed worker
25   Pid = spawn(worker,proc,[WorkerState]),
26
27   % Add the new worker to the supervisor state
28   % storage
29   NewInternalState =replace(From,Pid,InternalState),
30
31   % Infinite loop
32   loop(NewInternalState);
33 end.
```

Listing 2: Regeneration of a Worker Process in Case of Failure

This represents the part of the main loop of the supervisor process dealing with the management of the failure of a worker. As soon as a worker error is received (line 19), the supervisor retrieves the last operational state of the worker that has stopped (line 22), releases all of its allocated resources (line 26) and recreates a new worker process with the operational state of the stopped process (line 31). The supervisor continually turns in *loop* while awaiting new messages (line 40). The *loop* function call (lines 17 and 40) is *tail recursive*, thereby guaranteeing that the supervision process will grow in a constant memory space.

3.2 DNS Resolution

Before contacting a Web server, the worker process needs to convert the Domain Name Server (DNS) into a valid IP address. Whereas other systems (Mercator, Internet Archive) are forced to set up DNS resolvers each time a new link is identified, this is not necessary with Dominos. Indeed, in the framework of French Web legal deposit, the sites to be archived have been identified beforehand, thus requiring only one DNS resolution per domain name. This considerably increases crawl speed. The sites concerned include all online newspapers, such as LeMonde (*http://www.lemonde.fr/*), LeFigaro (*http://www.lefigaro.fr/*) . . . , online television/radio such as TF1 (*http://www.tf1.fr/*), M6 (*http://www.m6.fr/*) . . .

4. DETAILS OF IMPLEMENTATION

4.1 Web Client

The workers are the medium responsible for physically crawling on-line contents. They provide a specialized *wrapper* around the *libCURL*[5] library that represents the heart of the HTTP client. Each worker is interfaced to libCURL by a C driver (shared library). As the system seeks maximum network accessibility (communication protocol support), libCURL appeared to be the most judicious choice when compared with other available libraries.[6].

The protocols supported include: FTP, FTPS, HTTP, HTTPS, LDAP, Certifications, Proxies, Tunneling etc.

Erlang's portability was a further factor favoring the choice of libCURL. Indeed, libCURL is available for various architectures: Solaris, BSD, Linux, HPUX, IRIX, AIX, Windows, Mac OS X, OpenVMS etc. Furthermore, it is fast, thread-safe and IPv6 compatible.

This choice also opens up a wide variety of functions. Redirections are accounted for and powerful filtering is possible according to the type of content downloaded, headers, and size (partial storage on RAM or disk depending on the document's size).

4.2 Document Fingerprint

For each download, the worker extracts the hypertext links included in the HTML documents and initiates a fingerprint (signature operation). A fast fingerprint (*HAVAL* on 256 bits) is calculated for the document's content itself so as to differentiate those with similar contents (e.g. mirror sites). This technique is not new and has already been used in Mercator[13]. It allows redundancies to be eliminated in the archive.

4.3 URL Extraction and Normalization

Unlike other systems that use libraries of regular expressions such as PCRE[7] for URL extraction, we have opted

[5]Available at *http://curl.haxx.se/libcurl/*
[6]See *http://curl.haxx.se/libcurl/competitors.html*
[7]Available at *http://www.pcre.org/*

for the *Flex* tool that definitely generates a faster parser. Flex was compiled using a 256Kb buffer in which all table compression options were activated during parsing ”-8 -f -Cf -Ca -Cr -i”. Our current parser analyzes around 3,000 pages/second for a single worker for an average 49Kb per page.

According to [20], a URL extraction speed of 300 pages/second may generate a list of more than 2,000 URLs on average. A naive representation of structures in the memory may soon saturate the system.

Various solutions have been proposed to alleviate this problem. The Internet Archive [4] crawler uses *Bloom filters* in random access memory. This makes it possible to have a compact representation of links retrieved, but also generates errors (false-positive), i.e. certain pages are never downloaded as they create collisions with other pages in the Bloom filter. Compression without loss may reduce the size of URLs to below $10Kb$ [2, 21], but this remains insufficient in the case of large-scale crawls. A more ingenious approach is to use persistent structures on disk coupled with a cache as in Mercator [13].

4.4 URL Caching

In order to speed up processing, we have developed a scalable cache structure for the research and storage of URLs already archived. Figure 1 describes how such a cache works:

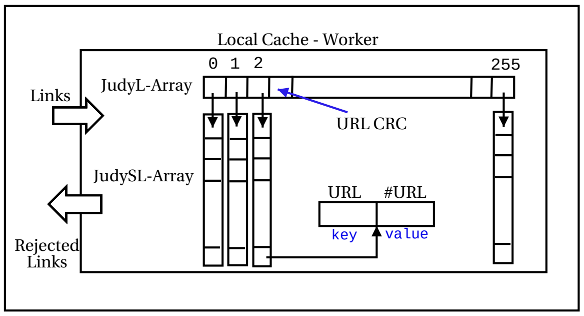

Figure 1: Scalable Cache

The cache is available at the level of each worker. It acts as a filter on URLs found and blocks those already encountered. The cache needs to be *scalable* to be able to deal with increasing loads. Rapid implementation using a non-reversible hash function such as HAVAL, TIGER, SHA1, GOST, MD5, RIPEMD ... would be fatal to the system's scalability. Although these functions ensure some degree of uniqueness in fingerprint constructionthey are too slow to be acceptable in these constructions. We cannot allow latency as far as lookup or URL insertion in the cache is concerned, if the cache is apt to exceed a certain size (over 10^7 key-value on average). This is why we have focused on the construction of a generic cache that allows key-value insertion and lookup in a scalable manner. The Judy-Array API[8] enabled us to achieve this objective. Without going into detail about Judy-Array (see their site for more information), our cache is a coherent coupling between a *JudyL-Array* and *N JudySL-Array*. The JudyL-Array represents a hash table of $N = 2^8$ or $N = 2^{16}$ buckets able to fit into the internal cache of the CPU. It is used to store ”*key-numeric value*” pairs where the key represents a CRC of the URL and whose value is a pointer to a JudySL-Array. The second, JudySL-Array, is a ”*key-compressed character string value*” type of hash, in which the key represents the URL identifier and whose value is the number of times that the URL has been viewed. This cache construction is completely scalable and makes it possible to have sub-linear response rates, or linear in the worst-case scenario (see Judy-Array at for an in-depth analysis of their performance). In the section on experimentation (section 5) we will see the results of this type of construction.

[8]Judy Array at the address: *http://judy.sourceforge.net/*

4.5 Limiting Disk Access

Our aim here is to eliminate random disk access completely. One simple idea used in [20] is periodically to switch structures requiring much memory over onto disk. For example, random access memory can be used to keep only those URLs found most recently or most frequently, in order to speed up comparisons. This requires no additional development and is what we have decided to use. The persistency of data on disk depends on the size of data in DS memory, and their DA age. The data in the memory are distributed transparently via Mnesia, specially designed for this kind of situation. Data may be duplicated ($\{ram\_copies, [Nodes]\}$, $\{disc\_copies, [Nodes]\}$) or fragmented ($\{frag\_properties, ...\}$) on the nodes in question.

According to [20], there are on average 8 non-duplicated hypertext links per page downloaded. This means that the number of pages retrieved and not yet archived is considerably increased. After archiving 20 million pages, over 100 million URLs would still be waiting. This has various repercussions, as newly-discovered URLs will be crawled only several days, or even weeks, later. Given this speed, the base's data refresh ability is directly affected.

4.6 High Availability

In order to apprehend the very notion of *High Availability*, we first need to tackle the differences that exist between a system's reliability and its availability. Reliability is an attribute that makes it possible to measure service continuity when no failure occurs.

Manufacturers generally provide a statistical estimation of this value for this equipment: we may use the term *MTBF* (Mean Time Between Failure). A strong MTBF provides a valuable indication of a component's ability to avoid overly frequent failure.

In the case of a complex system (that can be broken down into hardware or software parts), we talk about *MTTF* (Mean Time To Failure). This denotes the average time elapsed until service stops as the result of failure in a component or software.

The attribute of availability is more difficult to calculate as it includes a system's ability to react correctly in case of failure in order to restart service as quickly as possible.

It is therefore necessary to quantify the time interval during which service is unavailable before being re-established: the acronym *MTTR* (Mean Time To Repair) is used to represent this value.

The formula used to calculate the rate of a system's availability is as follows:

$$availability = \frac{MTTF}{MTTF + MTTR} \tag{4}$$

A system that looks to have a high level of availability should have either a strong MTTF, or a weak MTTR.

Another more practical approach consists in measuring the time period during which service is down in order to evaluate the level of availability. This is the method most frequently adopted, even if it fails to take account of the frequency of failure, focusing rather on its duration.

Calculation is usually based on a calendar year. The higher the percentage of service availability, the nearer it comes to High Availability.

It is fairly easy to qualify the level of High Availability of a service from the cumulated downtime, by using the normalized principle of "*9's*" (below 3 nine, we are no longer talking about High Availability, but merely availability). In order to provide an estimation of Dominos' High Availability, we carried out performance tests by *fault injection*. It is clear that a more accurate way of measuring this criterion would be to let the system run for a whole year as explained above. However, time constraints led us to adopt this solution. Our injector consists in placing pieces of false code in each part of the system and then measuring the time required for the system to make the service available. Once again, Erlang has proved to be an excellent choice for the setting up of these regression tests. The table below shows the average time required by Dominos to respond to these cases of service unavailability.

Table 1 clearly shows Dominos' High Availability. We

Service	Error	MTTR (microsec)
GenServer	10^3 bad match	320
cRPC	10^3 bad match	70
DDB	10^7 tuples	$9 * 10^6$
Node	10^3 bad match	250
Supervisor	10^3 bad match	60
Worker	10^3 bad match	115

Table 1: MTTR Dominos

see that for 10^3 matches of error, the system resumes service virtually instantaneously. The DB was tested on 10^7 tuples in random access memory and resumed service after approximately 9 seconds. This corresponds to an excellent MTTR, given that the injections were made on a PIII-966Mhz with 512Mb of RAM. From these results, we may label our system as being High Availability, as opposed to other architectures that consider High Availability only in the sense of failure not affecting other components of the system, but in which service restart of a component unfortunately requires manual intervention every time.

5. EXPERIMENTATION

This section describes Dominos' experimental results tested on 5 DELL machines:

- *nico*: Intel Pentium 4 - 1.6 Ghz, 256 Mb RAM. Crawl node (supervisor, workers). Activates a local cRPC.
- *zico*: Intel Pentium 4 - 1.6 Ghz, 256 Mb RAM. Crawl node (supervisor, workers). Activates a local cRPC.
- *chopin*: Intel Pentium 3 - 966 Mhz, 512 Mb RAM. Main node loaded on ServerGen and DB. Also handles crawling (supervisor, workers). Activates a local cRPC.
- *gao*: Intel Pentium 3 - 500 Mhz, 256 Mb RAM. Node for DB fragmentation. Activates a local cRPC.
- *margo*: Intel Pentium 2 - 333 Mhz, 256 Mb RAM. Node for DB fragmentation. Activates a local cRPC.

Machines chopin, gao and margo are not dedicated solely to crawling and are used as everyday workstations. Disk size is not taken into account as no data were actually stored during these tests. Everything was therefore carried out using random access memory with a network of 100 Mb/second. Dominos performed 25,116,487 HTTP requests after 9 hours of crawling with an average of 816 documents/second for 49Kb per document. Three nodes (nico, zico and chopin) were used in crawling, each having 400 workers. We restricted ourselves to a total of 1,200 workers, due to problems generated by Dominos at intranet level. The firewall set up to filter access is considerably detrimental to performance because of its inability to keep up with the load imposed by Dominos. Third-party tests have shown that peaks of only 4,000 HTTP requests/second cause the immediate collapse of the firewall. The firewall is not the only limiting factor, as the same tests have shown the incapacity of Web servers such as Apache2, Caudium or Jigsaw to withstand such loads (see *http://www.sics.se/~joe/apachevsyaws.html*). Figure 2 (left part) shows the average URL extraction per document crawled using a single worker. The abscissa (x) axis represents the number of documents treated, and the ordered (y) axis gives the time in microseconds corresponding to extraction. In the right-hand figure, the abscissa axis represents the same quantity, though this time in terms of data volume (Mb). We can see a high level of parsing reaching an average of 3,000 pages/second at a speed of 70Mb/second. In Figure 3 we see that URL normalization

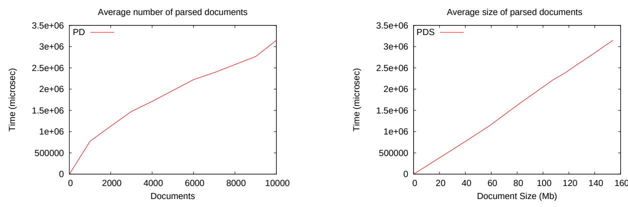

Figure 2: Link Extraction

is as efficient as extraction in terms of speed. The abscissa axis at the top (and respectively at the bottom) represents the number of documents processed per normalization phase (respectively the quantity of documents in terms of volume). Each worker normalizes on average 1,000 documents/second, which is equivalent to 37,000 URLs/second at a speed of 40Mb/second. Finally, the URL cache structure ensures a high degree of scalability (Figure 3). The abscissa axis in this figure represents the number of key-values inserted or retrieved. The cache is very close to a step function due to key compression in the Judy-Array. Following an increase in insertion/retrieval time in the cache, it appears to plateau by 100,000 key-value bands. We should however point out that URL extraction and normalization also makes use of this type of cache so as to avoid processing a URL already encountered.

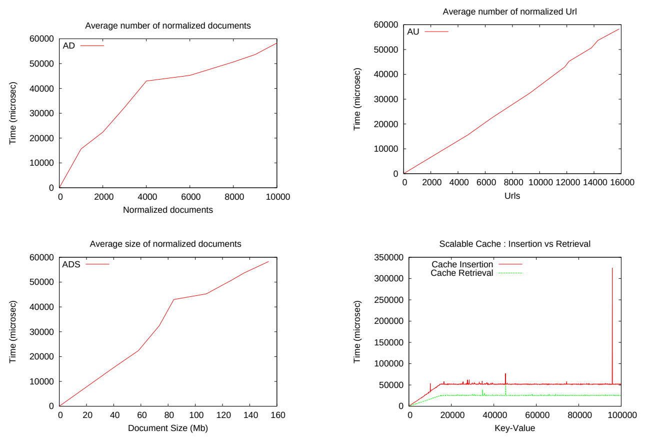

Figure 3: URL Normalization and Cache Performance

6. CONCLUSION

In the present paper, we have introduced a high availability system of crawling called Dominos. This system has been created in the framework of experimentation for French Web legal deposit carried out at the Institut National de l'Audiovisuel (INA). Dominos is a dynamic system, whereby the processes making up its kernel are mobile. 90% of this system was developed using Erlang programming language, which accounts for its highly flexible deployment, maintainability and enhanced fault tolerance. Despite having different objectives, we have been able to compare it with other documented Web crawling systems (Mercator, InternetArchive ...) and have shown it to be superior in terms of crawl speed, document parsing and process management without system restart.

Dominos is more complex than its description here. We have not touched upon archival storage and indexation. We have preferred to concentrate rather on the detail of implementation of the Dominos kernel itself, a strategic component that is often overlooked by other systems (in particular those that are proprietary, others being inefficient).

However, there is still room for the system's improvement. At present, crawled archives are managed by NFS, a file system that is moderately efficient for this type of problem. Switchover to Lustre[9], a distributed file system with a radically higher level of performance, is underway.

7. REFERENCES

[1] Z. BarYossef, A. Berg, S. Chien, J. Fakcharoenphol, and D. Weitz. Approximating aggregate queries about web pages via random walks. *In Proc. of 26th Int. Conf. on Very Large Data Bases*, 2000.

[2] K. Bharat, A. Broder, M. Henzinger, P. Kumar, and S. Venkatasubramanian. The connectivity server: Fast access to linkage. *Information on the Web*, 1998.

[3] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *In Proc. of the Seventh World-Wide Web Conference*, 1998.

[4] M. Burner. Crawling towards eternity: Building an archive of the world wide web. *http://www.webtechniques.com/archives/1997/05/burner/*, 1997.

[5] S. Chakrabarti, M. V. D. Berg, and B. Dom. Distributed hypertext resource discovery through example. *In Proc. of 25th Int. Conf. on Very Large Data Base*, pages 375–386, 1997.

[6] S. Chakrabarti, M. V. D. Berg, and B. Dom. Focused crawling: A new approach to topic-specific web resource discovery. *In Proc. of the 8th Int. World Wide Web Conference*, 1999.

[7] J. Cho and H. Garcia-Molina. The evolution of the web and implications for an incremental crawler. *In Proc. of 26th Int. Conf. on Very Large Data Bases*, pages 117–128, 2000.

[8] J. Cho and H. Garcia-Molina. Synchronizing a database to improve freshness. *In Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, 2000.

[9] J. Cho, H. Garcia-Molina, and L. Page. Efficient crawling through url ordering. *In 7th Int. World Wide Web Conference*, 1998.

[10] M. Diligenti, F. Coetzee, S. Lawrence, C. Giles, and M. Gori. Focused crawling using context graphs. *In Proc. of 26th Int. Conf. on Very Large Data Bases*, 2000.

[11] M. Henzinger, A. Heydon, M. Mitzenmacher, and M. Najork. Measuring index quality using random walks on the web. *In Proc. of the 8th Int. World Wide Web Conference (WWW8)*, pages 213–225, 1999.

[12] M. Henzinger, A. Heydon, M. Mitzenmacher, and M. Najork. On near-uniform url sampling. *In Proc. of the 9th Int. World Wide Web Conference*, 2000.

[13] A. Heydon and M. Najork. Mercator: A scalable, extensible web crawler. *World Wide Web Conference*, pages 219–229, 1999.

[14] J. Hirai, S. Raghavan, H. Garcia-Molina, and A. Paepcke. Webbase : : A repository of web pages. *In Proc. of the 9th Int. World Wide Web Conference*, 2000.

[15] B. Kahle. Archiving the internet. *Scientific American*, 1997.

[16] S. Lawrence and C. L. Giles. Searching the world wide web. *Science 280*, pages 98–100, 1998.

[17] M. Najork and J. Wiener. Breadth-first search crawling yields high-quality pages. *In 10th Int. World Wide Web Conference*, 2001.

[18] J. Rennie and A. McCallum. Using reinforcement learning to spider the web efficiently. *In Proc. of the Int. Conf. on Machine Learning*, 1999.

[19] S. Russel and P. Norvig. *Artificial Intelligence: A modern Approach*. Prentice Hall, 1995.

[20] V. Shkapenyuk and T. Suel. Design and implementation of a high-performance distributed web crawler. *Polytechnic University: Brooklyn*, Mars 2001.

[21] T. Suel and J. Yuan. Compressing the graph structure of the web. *In Proc. of the IEEE Data Compression Conference*, 2001.

[22] J. Talim, Z. Liu, P. Nain, and E. Coffman. Controlling robots of web search engines. *In SIGMETRICS Conference*, 2001.

[9] *http://www.lustre.org/*