

# High-Performance Crawling and Filtering in Java

Demetris Zeinalipour-Yazti<sup>1</sup> \* Marios Dikaiakos<sup>2</sup>

<sup>1</sup> WinMob Technologies Ltd., Agiou Antoniou 5,  
P.O. Box 20922, CYPRUS

<sup>2</sup> Dept. of Computer Science University of Cyprus, 75 Kallipoleos Street,  
PO Box 20537, CYPRUS

**Abstract.** In paper, we study the employment of crawlers as a programmable, scalable, and distributed component in future Internet middleware infrastructures and proxy services. In particular, we present the architecture and implementation of, and experimentation with WebRACE, a high-performance, distributed Web crawler, filtering server and object cache. We address the challenge of designing and implementing modular, open, distributed, and scalable crawlers, using Java. We describe our design and implementation decisions, and various optimizations. We discuss the advantages and disadvantages of using Java to implement the WebRACE-crawler, and present an evaluation of its performance.

## 1 Introduction

In this paper we present the architecture and implementation of, and early experimentation with WebRACE, a prototype HTTP Retrieval, Annotation and Caching Engine. WebRACE is part of a more generic system, called *eRACE* (extensible Retrieval, Annotation and Caching Engine), which is a distributed middleware infrastructure that enables the development and deployment of information dissemination services on Internet. eRACE services collect information from heterogeneous Internet sources according to pre-registered, XML-encoded user profiles. These profiles drive the collection of information and determine the relevance and the urgency of collected information. eRACE offers a functionality that goes beyond the capabilities of traditional Web servers and proxies, providing support for intelligent personalization, customization and transcoding of content, to match the interests and priorities of individual end-users through fixed and mobile terminals. It enables the development of new services and the easy re-targetting of existing services to new terminal devices.

WebRACE is the Web-specific proxy of eRACE. It crawls the Web to retrieve documents according to user profiles. The system subsequently caches and processes retrieved documents. Processing is guided by pre-defined user queries and consists of keywords-searches, title-extraction, summarizing, classification based on relevance with respect to user-queries, estimation of priority, urgency, etc. WebRACE processing results are encoded in a WebRACE-XML grammar and fed into a dissemination server,

---

\* Research supported partly by the grant PENEK-No 23/2000 from the Research Promotion Foundation of Cyprus, and by WinMob Technologies LTD, Nicosia, Cyprus

which selects dynamically among a suite of available choices for information dissemination, such as “push” vs. “pull,” the formatting and transcoding of data (HTML, WML, XML), the connection modality (wireless vs. wire-based), the communication protocol employed (HTTP, GSM/WAP, SMS), etc.

In this paper we describe our implementation experience with using Java to develop the high-performance Crawler, Annotation Engine and Object Cache of WebRACE. We also describe a number of techniques employed to achieve high-performance, such as distributed design to enable the execution of crawler modules to different machines, support for multithreading, customized memory management, employment of persistent data structures with disk-caching support, optimizations of the Java core libraries for TCP/IP and HTTP communication, etc.

## 2 WebRACE Design and Implementation Challenges

Two basic components comprises WebRACE, the *Mini-crawler* and the *Annotation Engine*, which operate independently and asynchronously (see Figure 1). Both components can be distributed to different computing nodes, execute in different Java heap spaces, and communicate through a permanent socket link; through this socket, the Mini-crawler notifies the Annotation Engine every time it fetches and caches a new page in the Object Cache. The Annotation Engine can then process the fetched page asynchronously, according to pre-registered user profiles or other criteria.

In the development of WebRACE we address a number of challenges. First is the design and implementation of a user-driven crawler. Typical crawlers employed by major search engines such as Google [1], start their crawls from a carefully chosen fixed set of “seed” URL’s. In contrast, the Mini-crawler of WebRACE receives continuously crawling directives which emanate from a queue of standing eRACE requests (see Figure 1). These requests change dynamically with shifting eRACE-user interests, updates in the base of registered users, changes in the set of monitored resources, etc.

Second, is the design of a crawler that monitors Web-sites exhibiting frequent updates of their content. WebRACE should follow and capture these updates so that interested users are notified by eRACE accordingly. Consequently, WebRACE is expected to crawl and index parts of the Web under short-term time constraints and keep multiple versions of the same Web-page in its store, until all interested users receive the corresponding alerts.

Similarly to personal and site-specific crawlers like SPHINX [2] and NetAttache Pro [3], WebRACE is customized and targets specific Web-sites. These features, however, must be sustained in the presence of a large and increasing user base, with varying interests and different service-level requirements. In this context, WebRACE must be scalable, sustaining high-performance and short turn-around times when serving many users and crawling a large portion of the Web. To this end, it should avoid duplication of effort and combine similar requests when serving similar user profiles. Furthermore, it should provide built-in support for QoS policies involving multiple service-levels and service-level guarantees. Consequently, the scheduling and performance requirements of WebRACE crawling and filtering face very different constraints than systems like Google [1], Mercator [5], SPHINX [2] or NetAttache Pro [3].

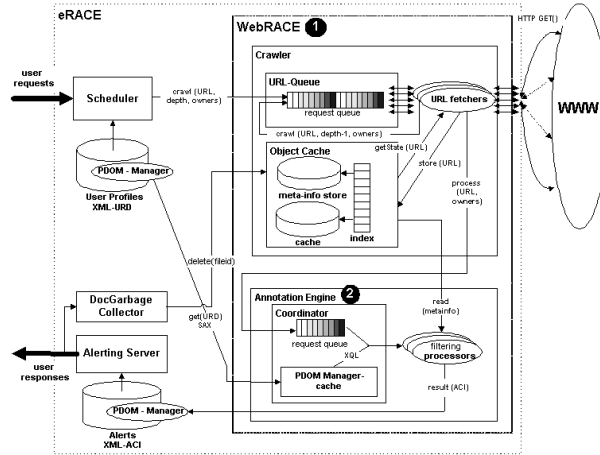


Fig. 1. WebRACE System Architecture.

### 3 SafeQueue: A High-performance Queue

At the core of WebRACE lies *SafeQueue*, a data-structure that we designed and implemented in Java to guarantee the efficient and robust operation of our agent-proxy. Queues are used in systems where the rate of incoming requests is larger than the rate of serviced requests, or where this relation is unknown in advance. Usually, Internet systems incorporate queues at the Application Layer to ensure that incoming requests will not be “lost” during periods of bursty load, system crashes, etc.

SafeQueue (SQ) is a typical FIFO queue used in a number of critical components of WebRACE; for example, WebRACE maintains its pending URL requests while crawling the Web and processing downloaded Web pages as Java objects in a SQ data structure. During a long crawl, millions of URL objects would have to be inserted and deleted from the queue. Consequently, an implementation of SafeQueue as a `java.util.LinkedList` component of Java [4] would result to an excessive number of expensive calls to object constructors, the continuous allocation and de-allocation of objects and an increased activity of the Garbage Collector, leading to performance degradation and frequent crashes due to heap-memory overflows. To overcome these problems, we implemented SafeQueue as a circular array of *QueueNode* objects with its own memory-management mechanism, which enables the re-use of objects and minimizes the garbage-collection overhead. Moreover, we incorporated support for persistence, overflow control, disk caching, multi-threaded access, and fast indexing to avoid the insertion of duplicate *QueueNode* entries (see Figure 2).

### 4 The URLFetcher

The *URLFetcher* is a WebRACE module that fetches a document from the Web when provided with a corresponding URL. The *URLFetcher* is implemented as a simple Java-thread, which supports both HTTP/1.0 [7] and HTTP/1.1 [8]. Similarly to crawlers

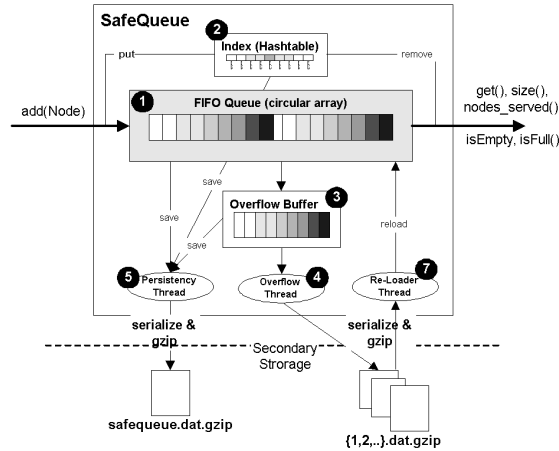


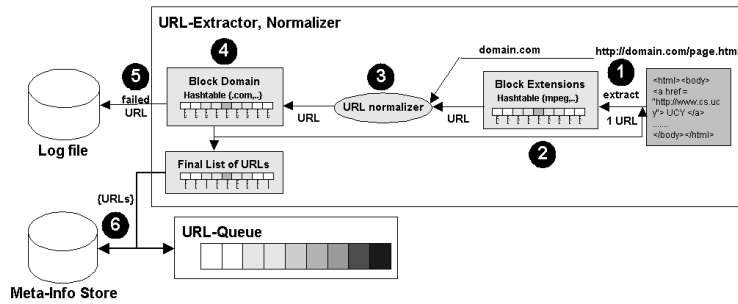
Fig. 2. SafeQueue Architecture.

like Mercator [5], WebRACE supports multiple URLFetcher threads running concurrently, grabbing pending requests from the URLQueue, conducting synchronous I/O to download WWW content, and overlapping I/O with computation. In the current version of WebRACE, resource management and thread scheduling is left to Java's runtime system and the underlying operating system.

The URLFetcher supports the Robots Exclusion Protocol (REP), which allows Web masters to declare parts of their sites off-limits to crawlers. In addition to supporting the standard Robots Exclusion Protocol, WebRACE supports the exclusion of particular domains and URL's.

In addition to handling HTTP connections, the URLFetcher processes the documents it downloads from the Web. To this end, it invokes methods of its *URLExtractor and normalizer* sub-component. The URLExtractor extracts links (URL's) out of a page, disregards URL's pointing to uninteresting resources, normalizes the URL's so that they are valid and absolute and, finally, adds these links to the URLQueue. The URL-extractor is exposed to all kinds of URL links that point to media types which may not be interesting for a particular, specialized crawl. As shown in Figure 3, the URLExtractor and normalizer works as a 6-step pipe within the URLFetcher. Extraction and normalization of URL's works as follows: in step 1, a *fastfind()* method identifies candidate URL's in the web-page at hand, removes internal links, and extracts the first URL that is candidate for processing. The efficient implementation of *fastfind* is challenging due to the abundance of badly formed HTML code on the Web. As an alternative solution we could reuse components such as Tidy [9] or its Java port, JTidy [10], to transform the downloaded Web page into well-formed HTML, and then extract all links using a generic XML parser. This solution proved to be too slow, in contrast to our *fastfind()* method which extracts links from a 70KB web page in approximately 80ms.

In step 2, a *Proactive Link Filtering* (PLF) method is invoked to disregard links to



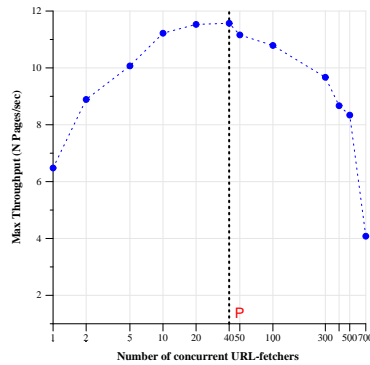
**Fig. 3.** URL Extractor Architecture

resources that are of no interest to the particular crawl. PLF uses the `/conf/ignore.types` configuration file of WebRACE to determine the file extensions that should be blocked during the URL extraction phase. Deciding if a link should be dropped takes less than *1ms* and saves WebRACE of the unnecessary effort to normalize a URL, add it to the URLQueue, and open an HTTP connection, just to see that this document has a media type that is not collected by the crawler.

Step 3 deals with the normalization of the URL at hand. To this end, we use our *URL-normalizer* method, which alters links that do not comply to the scheme-specific syntax of HTTP URL's, as defined in the HTTP RFCs. The URL-normalizer applies a set of heuristic corrections, which give on the average a 95% of valid and normalized URL's. The URL-normalizer took on the average *200ms* for 100 URL's.

Step 4 filters out links that belong to domains that are blocked or excluded by the Robot Exclusion Protocols. Steps 1 through 4 are executed repeatedly until all links of the document at hand have been processed. Step 5 logs the URL's that failed the normalization process for debugging purposes. Finally, at step 6, all extracted and normalized URL's are collectively added to the URL-Queue and stored to the Meta-Info Store. Caution is taken to drop duplicate URL's.

The URL extraction and normalization pipe requires an average of *300ms* to extract the links from a *70KB* HTML page and to normalize them appropriately, when executed on our Sun Enterprise E250 Server. To evaluate the overall performance of the URLFetcher, we ran a number of experiments, launching many concurrent fetchers that try to establish TCP connections and fetch documents from Web servers located on our 10/100Mbits LAN. Each URLFetcher pre-allocates all of its required resources before the benchmark start-up. The benchmarks ran on a 360MHz UltraSPARC-III, with 128MB RAM and Solaris 5.7. As we can see from Figure 4, the throughput increases with the number of concurrent URLFetchers, until a peak *P* is reached. After that point, throughput drops substantially. This crawling process took a very short time (3 minutes with only one thread), which is actually the reason why the peak value *P* is 40. In this case, URLQueue empties very fast, limiting the utilization of URLFetcher's near the benchmark's end. Running the same benchmark for a lengthy crawl we observed that 100 concurrent URLFetcher's achieve optimal crawling throughput.



Number of Concurrent URL-fetchers executing in WebRACE (normal-log scale)

**Fig. 4.** URL-fetcher throughput degradation.

## 5 The Object Cache

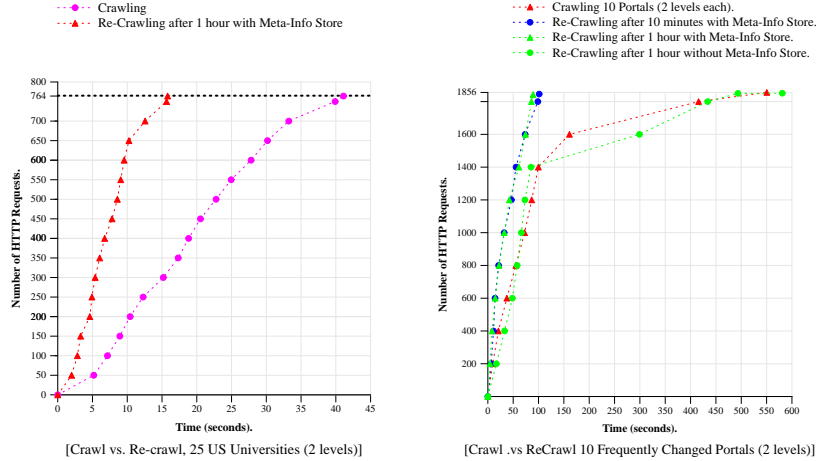
The *Object Cache* is the component responsible for managing documents cached in secondary storage. It is used for storing downloaded documents that will be retrieved later for processing, annotation and subsequent dissemination to eRACE users. The Object Cache, moreover, caches the crawling state in order to coalesce similar crawling requests and to accelerate the re-crawling of WWW resources that have not changed since their last crawl.

An *Index*, a *Meta-Info Store* and an *Object Store* (see Figure 1), comprises the Object Cache. Index resides in main memory and indexes documents stored on disk; it is implemented as a `java.util.HashMap`, which contains URL's that have been fetched and stored in WebRACE. That way, *URLFetcher's* can check if a page has been re-fetched, before deciding whether to download its contents from the Web. The Meta-Info Store collects and maintains meta-information for cached documents. Finally, the Object Store is a directory in secondary storage that contains a compressed version of downloaded resources.

### 5.1 Meta-Info Store

The Meta-Info Store maintains a meta-information file for each Web document stored in the Object Cache. Furthermore, a key for each meta-info file is kept with the Index of the Object Cache to allow for fast look-ups. The contents of a meta-info file are encoded in XML and include:

- The URL address of the corresponding document;
- The IP address of its origin Web server;
- The document size in KiloBytes;
- The Last-Modified field returned by the HTTP protocol during download;
- The HTTP response header, and all extracted and normalized links contained in this document.



**Fig. 5.** Crawling vs. re-crawling in WebRACE.

To avoid the overhead of the repeated downloading and analysis of documents that have not changed, we use the Meta-Info Store to decide whether to download a document that is already cached in WebRACE. More specifically:

3. If a document is in the cache:
  - Load its meta-info file and extract the HTTP Last-Modified time-stamp assigned by the origin server. Open a socket connection to the origin server and request the document using a conditional *HTTP GET* command (*if-modified-then*), with the extracted time-stamp as its parameter.
  - If the origin server returns a “304 (not modified)” response and no message-body, terminate the fetching of this particular resource, extract the document links from its meta-info file, and proceed to step 8.
  - Otherwise, download the body of the document, store it in main memory and proceed to step 4.

If a cached document has not been changed during a re-crawl, the URLFetcher proceeds with crawling the document’s outgoing links, which are stored in the Meta-Info Store, and which may have changed.

To assess the performance improvement provided by the use of the Meta-Info Store, we conducted an experiment with crawling two classes of Web sites. The first class includes servers that provide content which does not change very frequently (University sites). The second class consists of popular news-sites, search-engine sites and portals (cnn.com, yahoo.com, msn.com, etc.). For these experiments we configured WebRACE to use 150 concurrent URLFetchers and ran it on our Sun Enterprise E250 Server, with the Annotation Processor running concurrently on a Sparc 5.

The diagram of Figure 5 (left) presents the progress of the crawl and re-crawl operations for the first class of sites. As we can see from this diagram, the employment of

the Meta-Info Store results to an almost three-fold improvement in the crawling performance. Moreover, it reduces substantially the network traffic and the Web-servers' load generated because of the crawl.

The diagram of Figure 5 (right) presents our measurements from the crawl and re-crawl operations for the second class of sites. Here, almost 10% of the 993 downloaded documents change between subsequent re-crawls. From this diagram we can easily see the performance advantage gained by using the Meta-Info Store to cache crawling meta-information.

## 6 The Annotation Engine (AE)

The Annotation Engine processes documents that have been downloaded and cached in the *Object Cache* of WebRACE. Its purpose is to “classify” collected content according to user-interests described in eRACE profiles. The meta-information produced by the processing of the Annotation Engine is stored in WebRACE as annotation linked to the cached content. Pages which are irrelevant to user-profiles are dropped from the cache.

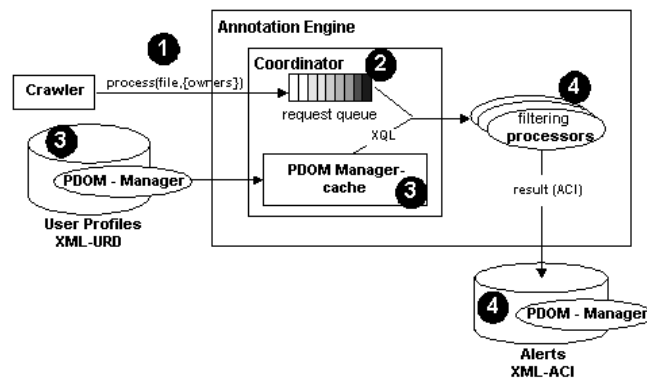


Fig. 6. WebRACE Annotation Engine.

Personalized annotation engines are not used in typical Search Engines [1], which employ general-purpose indices instead. To avoid the overhead of incorporating a generic look-up index in WebRACE that will be updated dynamically as resources are downloaded from the Web, we designed the AE so that it processes “on the fly” downloaded pages. Therefore, each time the Annotation Engine receives a ‘‘`process(file, {users})`’’ request through the established socket connection with the Mini-crawler, it inserts the request in the *Coordinator*, which is a SafeQueue data structure (see Figure 6). Multiple *Filtering Processors* remove requests from the Coordinator and process them according to the *Unified Resource Descriptions (URD’s)*[21] of eRACE users contained in the request. URD is an XML-encoded data structure that encapsulates source information, processing directives and urgency information for Web services monitored

by eRACE. Currently, the annotation engine implements a simple pattern-matching algorithm looking for weighted keywords that are included in the user-profiles.

### 6.1 Filtering Processor

Filtering Processor (FP) is the component responsible for evaluating if a document matches the interests of a particular eRACE-user, and for generating an ACI out of a crawled page. ACI [21] stands for Annotated Cached Information and is an extensible data structure that encapsulates information such as the (relevance, urgency, file size, e.t.c) that is generated after a resource is processed by the Filtering Processor. The Filtering Processor works as a pipe of filters: At step 1, FP loads and decompresses the appropriate file from the Object Cache of WebRACE. At step 2, it removes all links contained in the document and proceeds to step 3, where all special HTML characters are also removed. At step 4, any remaining text is added to a Keyword HashTable. Finally, at step 5, a pattern-matching mechanism loads sequentially all the required URD elements from the URD-PDOM and generates ACI meta-information, which is stored in the ACI-PDOM (step 6). This pipe requires an average of 200 msec to calculate the ACI for a *70KB* Web page, with 3 potential recipients.

## 7 Conclusions and Future Work

In this paper, we presented WebRACE, a World-Wide Web “agent-proxy” that collects, filters and caches Web documents. WebRACE is designed in the context of eRACE, an extensible Retrieval Annotation Caching Engine. eRACE collects, annotates and disseminates information from heterogeneous Internet sources and protocols (Web, email, newsgroups), according to XML-encoded user profiles that determine the urgency and relevance of collected information. The main component of WebRACE is a high-performance, distributed Web crawler and filtering processor, written entirely in Java. Although a number of papers have been published on Web crawlers [2, 5, 15, 16, 17], proxy services [19], information dissemination systems [20] and Internet middleware [18], the issue of incorporating flexible, scalable and user-driven crawlers in middleware infrastructures remains open. Furthermore, the adoption of Java as the language of choice in the design of Internet middleware and servers raises many doubts, primarily because of performance and scalability questions. There is no question, however, that Web crawlers written in Java will be an important component of such systems, along with modules that process collected content.

In our work, we address the challenge of designing and implementing a modular, user-driven, open, distributed, and scalable crawler and filtering processor, in the context of the eRACE middleware. We describe our design and implementation decisions, and various optimizations. Furthermore, we discuss the advantages and disadvantages of using Java to implement the crawler, and present an evaluation of its performance. To assess WebRACE’s performance and robustness we ran numerous experiments and crawls; several of our crawls lasted for days. Our system worked efficiently and with no failures when crawling local Webs in our LAN and University WAN, and the global Internet. Our experiments showed that our implementation is robust and reliable.

## References

1. S. Brin and L. Page : The Anatomy of a Large-Scale Hypertextual (Web) Search Engine, Computer Networks and ISDN Systems volume 30, number 1-7, pages 107-117, 1998
2. R. Miller and K. Bharat : SPHINX: A Framework for Creating Personal, Site-specific Web Crawlers, Proceedings of the Seventh International WWW Conference, pages 161-172, April 1998
3. Tympani Development Inc. : NetAttache Pro, 2000
4. J. Gosling and B. Joy and G. Steele, The Java Language Specification, Addison-Wesley, 1996
5. A. Heydon and M. Najork : Mercator: A Scalable, Extensible Web Crawler, World Wide Web, number 4, volume 2, pages 219-229, December,1999
6. M. Dikaiakos, D. Gunopoulos : FIGI: The Architecture of an Internet-based Financial Information Gathering Infrastructure. In Proceedings of the International Workshop on Advanced Issues of E-Commerce and Web-based Information Systems, pages 91-94, IEEE-Computer Society, April 1999
7. T. Berners-Lee and R. Fielding and H. Frystyk : Hypertext Transfer protocol – HTTP/1.0, W3C May 1996, <http://www.w3.org/Protocols/HTTP/1.0/spec.html>
8. J. Gettys and J. Mogul and H. Frystyk and L. Masinter and P. Leach and T. Berners-Lee : Hypertext Transfer protocol – HTTP/1.1, W3C June 1999, <http://www.w3.org/Protocols/rfc2616/rfc2616.html>
9. D. Raggett : Clean up your Web pages with HTML TIDY, <http://www.w3.org/People/Raggett/tidy/>
10. S. Lempinen : Jtidy, <http://lempinen.net/sami/jtidy>
11. G. Huck and I. Macherius and P. Fankhauser : PDOM: Lightweight Persistency Support for the Document Object Model, Proceedings of the 1999 OOPSLA Workshop Java and Databases: Persistence Options.(OOPSLA '99), ACM, SIGPLAN, November 1999, USA
12. GMD-IPSI XQL Engine, <http://xml.darmstadt.gmd.de/xql/>
13. Document Object Model (DOM) Level 1 Specification, W3C Recommendation 1, October 1998, <http://www.w3.org/TR/REC-DOM-Level-1/>
14. D. Zeinalipour-Yazti : eRACE: an eXtensible Retrieval, Annotation and Caching Engine, Department of Computer Science, University of Cyprus, B.Sc. Thesis. In Greek, June 2000
15. J. Cho and H. Garcia-Molina and L. Page : Efficient crawling through URL ordering, Proceedings of the Seventh International WWW Conference, pages 161-172, April 1998
16. S. Chakrabarti and M. van den Berg and B. Dom : Focused Crawling: A New Approach to Topic-Specific Web Resource Discovery, 8th World Wide Web Conference Toronto, May 1999
17. S. Raghavan and H. Garcia-Molina : Crawling the Hidden Web, VLDB 2001: 27th International Conference on Very Large Data Bases September 2001
18. S. Gribble et al. : The Ninja Architecture for Robust Internet-Scale Systems and Services, To appear in a Special Issue of Computer Networks on Pervasive Computing, 2001
19. C. M. Bowman and P. B. Danzig and D. R. Hardy and U. Manber and M. F. Schwartz : The Harvest Information Discovery and Access System, roceedings of the Second International WWW Conference 1995
20. T. W. Yan and H. Garcia-Molina : SIFT - A Tool for Wide-Area Information Dissemination, Proceedings of the 1995 USENIX Technical Conference pages 177-186, 1995
21. D. Zeinalipour-Yazti, M. Dikaiakos: High-Performance Crawling and Filtering in Java, Technical Report TR-2001-3, Department of Computer Science, University of Cyprus, June 2001