High-Performance Crawling and Filtering in Java

Demetris Zeinalipour-Yazti
WinMob Technologies Ltd.
P.O. Box 20922
Nicosia, Cyprus
csyiazti@ucy.ac.cy

Marios Dikaiakos
Dept. of Computer Science
University of Cyprus
PO Box 20537, Nicosia, Cyprus
mdd@ucy.ac.cy

Abstract

Web crawlers are the key component of services running on Internet and providing searching and indexing support for the entire Web, for corporate Intranets and large portal sites. More recently, crawlers have also been used as tools to conduct focused Web searches and to gather data about the characteristics of the WWW. 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. WebRACE is designed in the context of eRACE, an extensible Retrieval Annotation Caching Engine, which collects, annotates and disseminates information from heterogeneous Internet sources and protocols, according to XML-encoded user profiles that determine the urgency and relevance of collected information.

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, 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.

The remaining of the paper is organized as follows: Section 2 presents and overview of the WebRACE system architecture and the challenges addressed in our work. Section 3 describes the Java implementation of a high-performance persistent queue used in a number of WebRACE components. Sections 4 and 5 describe the design and implementation of a Crawler and Object Cache, used to retrieve and store content from the Web. Section 6 presents the Filtering Processor that analyzes the collected information, according to user-profiles. Finally, we conclude in Section 7 with conclusions and future work.

2 WebRACE Design and Implementation Challenges

The eRACE infrastructure consists of protocol-specific *Agent-Proxies*, like mailRACE, newsRACE and dbRACE, that gather information from POP3 email-accounts, USENET NNTP-news, and Web-database queries, respectively. WebRACE is the Agent-Proxy of eRACE that collects, processes and caches content from information sources on the WWW, accessible through the HTTP protocols (HTTP/1.0, HTTP/1.1), according to eRACE user-profiles. Other eRACE proxies have the same general architecture with WebRACE, differing only in the implementation of their protocol-specific

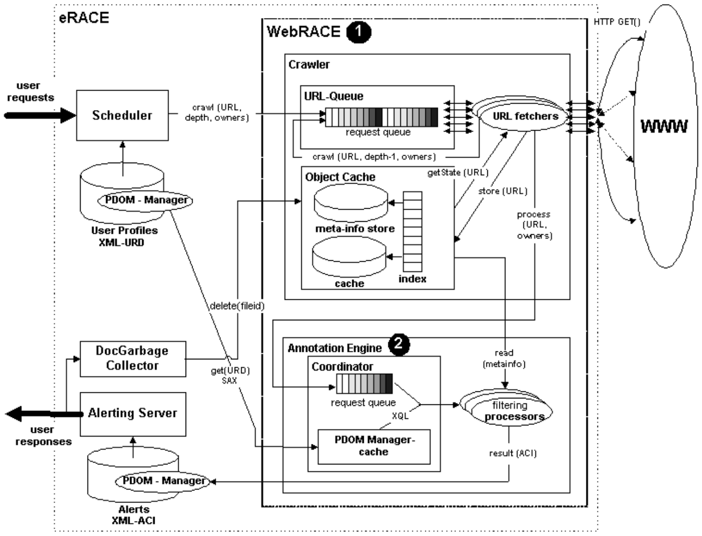


Figure 1: WebRACE System Architecture.

proxy engines.

WebRACE is comprised of two basic components, 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 [5], 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 [20] and NetAttache Pro [16], 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 [5], Mercator [14], SPHINX [20] or NetAttache Pro [16].

Finally, WebRACE is implemented entirely in Java [11]. Its implementation consists of approximately 5500 lines of code, 2649 of which correspond to the Mini-crawler implementation, 1184 to the Annotation Engine, 367 to the SafeQueue data structure, and 1300 to common I/O libraries. Java was chosen for a variety of reasons. Its object-oriented design enhances the software development process, supports rapid prototyping and enables the re-use and easy integration of existing components. Java class libraries provide support for key features of WebRACE: platform independence, multithreading, network programming, high-level programming of distributed applications, string processing, code mobility, compression, etc. Other Java features, such as automatic garbage collection, persistence and exception handling, are crucial in making our system more tolerant to run-time faults.

The choice of Java, however, comes with a certain risk-factor that arises from known performance problems of this programming language and its run-time environment. Notably, performance and robustness are issues of critical importance for a system like WebRACE, which is expected to function as a server, to run continuously and to sustain high-loads at short periods of time. In our experiments, we found the performance of Java SDK 1.3 satisfactory when used in combination with the Java HotSpot Server VM [19, 18]. Furthermore, the Garbage Collector, which seemed to be a problem with earlier Java versions, has a substantially improved performance and effectiveness under Java v.1.3.

Numerous experiments with earlier versions of WebRACE, however, showed that memory management cannot rely entirely on Java's garbage collection. During long crawls, memory allocation increased with crawl size and duration, leading to over-allocation of heap space, heap-space overflow exceptions, and system crashes. Extensive performance and memory debugging with the OptimizeIt profiler [25] identified a number of Java core classes that allocated new objects excessively and caused heap-space overflows and performance degradation. Consequently, we had to develop our own data-structures that use a bounded amount of heap-space regardless of the crawl size, and maintain part of their data on disk. Furthermore, we re-wrote some of the mission-critical Java classes, streamlining very frequent operations. More details are given in the sections that follow.

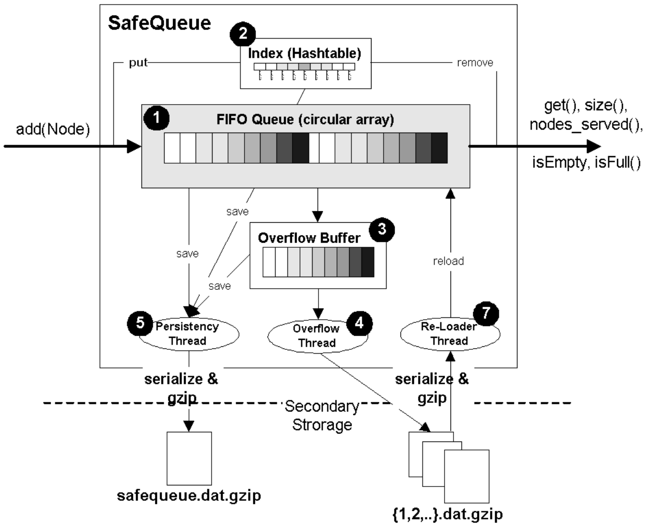


Figure 2: SafeQueue Architecture.

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.LinkList` component of Java [11] 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).

All memory required for the SafeQueue structure is bounded and pre-allocated during initialization; no new QueueNode objects are allocated or discarded during the execution of WebRACE. This is achieved with the implementation of a `reset()` method in the QueueNode class, which cleans the various objects contained in a QueueNode object, without de-allocating the object itself from the heap. SafeQueue implements a variety of blocking (`get(), add()`) and non-blocking methods (`isFull(), isEmpty(), nodesServed()`), which provide a programmer with transparent access to data located in the queue. The `add()` method makes sure that the queue is not full and assigns data to the first available QueueNode. The `get()` method returns the contents of SafeQueue's head and releases the corresponding object.

SafeQueue implements an overflow-management mechanism as follows: if the queue is full when an `add()` request is issued, SafeQueue withholds and returns the first available QueueNode of the *OverflowBuffer* (see Figure 2, point 3). As soon as this buffer is filled, its contents are compressed, serialized and flushed to secondary storage by the *OverflowThread*. This thread maintains a counter that is incremented every time a buffer is flushed to disk to provide unique names to stored buffers. Whenever QueueNodes are relinquished, the *ReloaderThread* is invoked and fetches QueueNode objects stored in overflow buffers and secondary storage (see Figure 2, point 7).

Many duplicate requests are generated during crawling because different Web pages often contain links to the same resource. SafeQueue's Index addresses this problem by ensuring that no two identical QueueNodes will be placed in SafeQueue. This mechanism is implemented with a `java.util.HashTable`, which indexes queued QueueNode's. Each time the `add()` method is called, the key of the respective QueueNode object is added to the SafeQueue Index. If the QueueNode key is already in the HashTable, the object is dropped. On the other hand, each time we invoke the `get()` method to remove an object from the queue, its key is also removed from the index.

Java-based systems running for long periods of times are exposed to system failures, Java Virtual Machine crashes, memory overflow exceptions, etc. Fault-tolerance in such systems is very important because a crawling process might require many days. SafeQueue provides persistence with the deployment of a *PersistencyThread*, which saves SafeQueue on secondary storage periodically and asynchronously, without blocking the operation of the Queue. In case of WebRACE failure, when the server restarts, it restores SafeQueue to its last saved state. This procedure is expensive because the PersistencyThread has to decompress and de-serialize the queue. The time required is always less than 1 minute for a Queue with 10^6 nodes. The interval of SafeQueue's storage is configurable through the server's settings.

4 The Mini-crawler of WebRACE

A crawler is a program that traverses the hypertext structure of the Web automatically, starting from an initial hyper-document and recursively retrieving all documents accessible from that document. Web crawlers are also referred to as robots, wanderers, or spiders. Typically, a crawler executes a basic algorithm that takes a list of “seed” URL’s as its input, and repeatedly executes the following steps [14]: It initializes the crawling engine with the list of seed URL’s and pops a URL out of the URL list. Then, it determines the IP address of the chosen URL’s host name, opens a socket connection to the corresponding server, asks for the particular document, parses the HTTP response header and decides if this particular document should be downloaded. If this is so, the crawler downloads the corresponding document and extracts the links contained in it; otherwise, it proceeds to the next URL. The crawler ensures that each extracted link corresponds to a valid and absolute URL, invoking a URL-normalizer to “de-relativize” it, if necessary. Then, the normalized URL is appended to the list of URL’s scheduled for download, provided this URL has not been fetched earlier.

In contrast to typical crawlers [20, 14], WebRACE refreshes continuously its URL-seed list from requests posted by the eRACE *Request Scheduler*. These requests have the following format:

`[Link, ParentLink, Depth, {owners}]`

Link is the URL address of the Web resource sought, *ParentLink* is the URL of the page that contained Link, *Depth* defines how deep the crawler should “dig” starting from the page defined by Link, and {*owners*} contains the list of eRACE users potentially interested in the page that will be downloaded.

The Mini-crawler is configurable through three files: a) `/conf/webrace.conf`, which contains general settings of the engine, such as the crawling start page, the depth of crawling, intervals between system-state save, the size of key data-structures maintained in main memory, etc.; b) `/conf/mime.types`, which controls what Internet media types should be gathered by the crawler; c) `/conf/ignore.types`, which controls what file extensions should be blocked by the engine; URL resources with a suffix listed in `ignore.types` will not be downloaded regardless of the actual mime-type of that file’s content. Making the Mini-crawler configurable through these configuration files renders it adaptable to specific crawl tasks and benchmarks. The crawling algorithm described in the previous section requires a number of components, which are listed and described in detail below:

- The *URLQueue* for storing links that remain to be downloaded.
- The *URLFetcher*, which downloads documents using the HTTP protocol. The URLFetcher contains also a *URL extractor and normalizer*, which extracts links from a document and ensures that the extracted links are valid and absolute URL’s.
- The *Object Cache*, which stores and indexes downloaded documents, and ensures that no duplicate documents are maintained in cache. The Object Cache, however, can maintain multiple versions of the same URL, if its contents have changed with time.

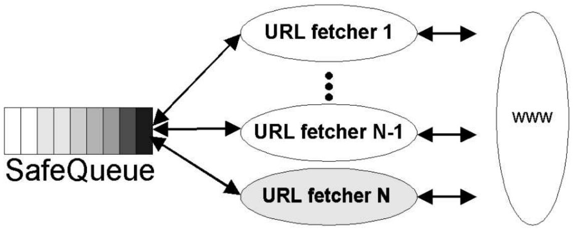


Figure 3: URL Fetchers.

4.1 The URLQueue

The *URLQueue* is an implementation of the SafeQueue data structure, comprised of URLQueueNode's. $URLQueueNode$'s are Java objects that capture requests coming from the Request Scheduler of eRACE. During the server's initialization, WebRACE allocates the full size of the URLQueue on the heap. The length of the URLQueue is determined during the server's initialization from WebRACE configuration files. At that time, our program allocates the heap-space required to store all the nodes of the queue. We chose this approach instead of allocating Queue Nodes on demand for memory efficiency and performance. In our experiments, we have configured the URLQueue size to two million nodes, i.e., two million URL's. This number corresponds to approximately $27MB$ of heap space. A larger URLQueue can be employed, however, at the expense of heap size available for other components of WebRACE. We are currently investigating ways to handle larger URLQueue sizes by making SafeQueue distributed [12].

4.2 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 [3] and HTTP/1.1 [10]. Similarly to crawlers like Mercator [14], 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 number of available URLFetcher threads, however, can be configured during the initialization of the WebRACE-server. It should be noted that a very large number of URLFetcher threads can lead to serious performance degradation of our system, due to excessive synchronization and context-switching overhead. In future work we plan to investigate schemes involving Java mobile agents to implement agile and self-adaptable fetchers [8].

The URLFetcher supports the Robots Exclusion Protocol (REP), which allows Web masters to declare parts of their sites off-limits to crawlers. The REP is driven by a text document located

```
# robots.txt for http://www.w3.org/
User-agent:  *
Disallow:  /Team
Disallow:  /Project
Disallow:  /Systems
Disallow:  /Web
Disallow:  /History
Disallow:  /Out-Of-Date
```

Table 1: Robot.txt file.

in the root of a Web Server, specifying which resources should not been accessed by crawlers. A typical Robot.txt file is shown in Table 1. In addition to supporting the standard Robots Exclusion Protocol, WebRACE supports the exclusion of particular domains and URL's. To implement the exclusion protocol, WebRACE provides a *BlockDomain* hash table, which contains all domains and URL's that should be blocked.

The URLFetcher uses the HTTP support provided by the JDK 1.2 Java class libraries, which enables the crawler to specify how long a socket can remain open "waiting" for the Web server to respond, through its `Socket.setSoTimeout()` method. In the current `java.net.Socket`-class implementation, however, socket objects are not reusable. Therefore, we had to modify the `java.net.Socket` implementation, adding an extra "`reset(String host, int port)`" method that enables the reuse of a socket object for a different host. Thus, we managed to reduce significantly the overhead of continuously constructing and destructing socket objects.

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* subcomponent. 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 4, 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 (starting from "#"), mailto links ("`mailto:`"), etc, 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 [23] or its Java port, JTidy [17], 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

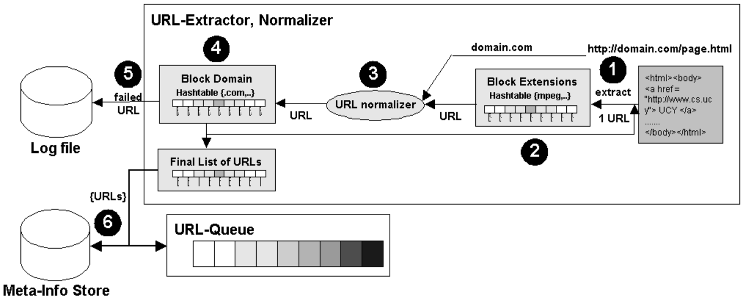


Figure 4: URL Extractor Architecture

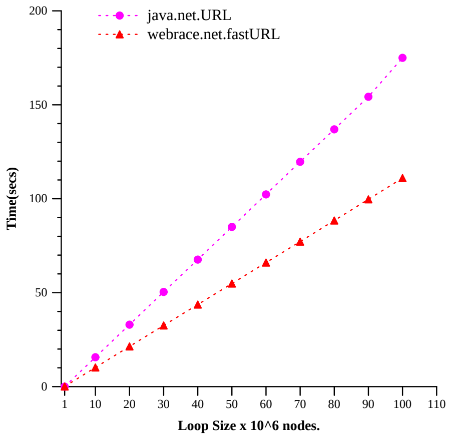
```
http_URL =   ‘‘http:’’  ‘‘//’’ host [ ‘‘:’’  port ] [ abs_path]
host =       < A legal Internet host domain name or IP address
             (in dotted-decimal form), as defined by Section 2.1 of RFC 1123 >
port =       *DIGIT
abs_path =   Absolute path of the resource starting from ‘‘/’’
```

Table 2: Valid URL Syntax.

`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 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 (see Table 2) [3, 10]. The URL-normalizer applies a set of heuristic corrections, which give on the average a 95% of valid and normalized URL's. For each Web page processed, the URL-normalizer made extensive use of the `java.net.URL` library while checking the syntactic validity of the normalized URL. Nevertheless, this library creates numerous objects that cannot be reused, resulting to excessive heap-memory consumption, an increased activity of the garbage collector, and significant performance degradation. Therefore, we implemented `webrace.net.fastURL`, a streamlined URL class that enables the reuse of URL objects via its `reparse()` method. This optimization achieves twofold and threefold improvements of the normalization performance under Solaris and Windows NT respectively. This can be seen from Figure 5, where we present the results of a `java.net.URL` vs. `webrace.net.fastURL` performance benchmark. In this benchmark, we evaluated `webrace.net.URL` by instantiating up to 10^8 new URL objects. The benchmark ran on a



Total Times:[java.net.URL .vs webrace.net.fastURL] Benchmark

Figure 5: webrace.net.URL Performance.

Sun Enterprise E250 Server with 2 UltraSPARC-II processors at 400MHz, with 512MB memory, running the Solaris 5.7 operating system. 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 6, 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.

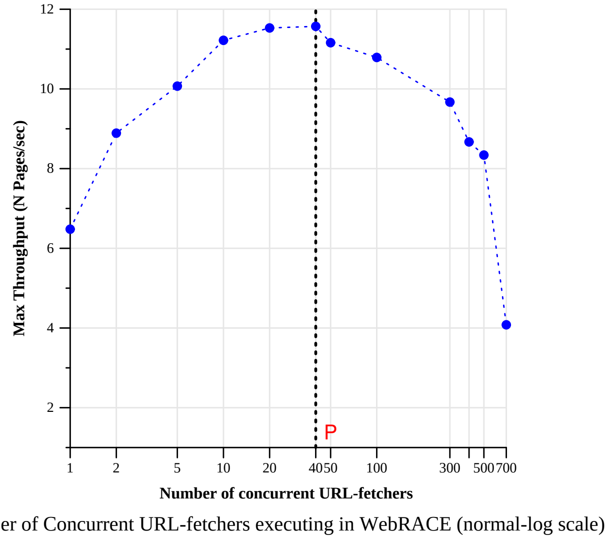


Figure 6: URL-fetcher throughput degradation.

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 coallesce similar crawling requests and to accelerate the re-crawling of WWW resources that have not changed since their last crawl.

The Object Cache is comprised of an *Index*, a *Meta-Info Store* and an *Object Store* (see Figure 1). The Index resides in main memory and indexes documents stored on disk; it is implemented as a `java.util.HashTable`, 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;

```
< webrace:url>http://www.cs.ucy.ac.cy/~epl121/< /webrace:url>
< webrace:ip>194.42.7.2< /webrace:ip>
< webrace:kbytes>1< /webrace:kbytes>
< webrace:ifmodifiedsince>989814504121< /webrace:ifmodifiedsince>
<webrace:header>
  HTTP/1.0 200 OK
  Server:  Netscape-FastTrack/2.01
  Date:  Fri, 11 May 2001 13:50:10 GMT
  Accept-ranges:  bytes
  Last-modified:  Fri, 26 Jan 2001 21:46:08 GMT
  Content-length:  1800
  Content-type:  text/html
< /webrace:header>
<webrace:links>
  http://www.cs.ucy.ac.cy/Computing/labs.html
  http://www.cs.ucy.ac.cy/
  http://www.cs.ucy.ac.cy/helpdesk
< /webrace:links>
```

Table 3: Example of meta-information file.

- 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.

An example of a meta-info file is given in Table 3. Meta-information is used to accelerate the re-crawling of visited Web sites as follows: Normally, a *URLFetcher* executes the following algorithm to download a Web page:

1. Retrieve a QueueNode from the URLQueue and extract its URL.
2. Retrieve the URL and analyze the HTTP-header of the response message. If the host server contains the message “200 Ok,” proceed to the next step. Otherwise, continue with the next QueueNode.
3. Download the body of the document and store it in main memory.
4. Extract and normalize all links contained in the downloaded document.
5. Compress and save the document in the Object Cache.

6. Save a generated meta-info file in the Meta-Info Store.

7. Add the key (`hashCode`) of the fetched URL to the Index of the Object Cache.

8. Notify the Annotation Engine that a new document has been fetched and stored in the Object Cache.

9. Add all extracted URL's to the URLQueue.

To avoid the overhead of the repeated downloading and analysis of documents that have not changed, we alter the above algorithm and use the Meta-Info Store to decide whether to download a document that is already cached in WebRACE. More specifically, we change the second and third steps of the above crawling algorithm as follows:

2. Access the Index of the Object Cache and check if the URL retrieved from the URLQueue corresponds to a document fetched earlier and cached in WebRACE.

3. If the document is not in the Cache, download it and proceed to step 4. Otherwise:

    - 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.

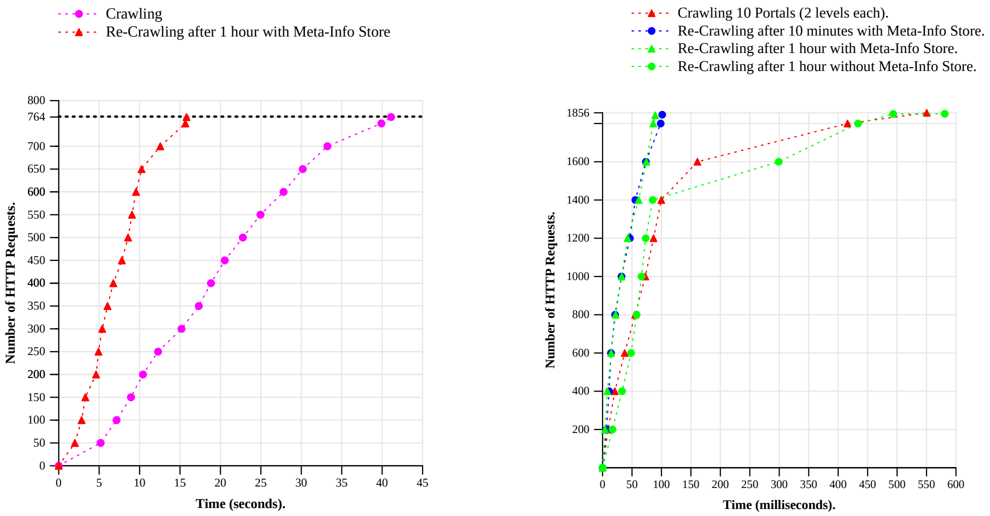
    - If the origin server returns a "`304 (not modified)`" response and no message-body, terminate the fetching of this particular resource, extract the document links from its meta-info file, and proceed to step 8.

    - 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 7 (left) presents the progress of the crawl and re-crawl operations for the first class of sites. The time interval between the crawl and the subsequent re-crawl was one hour; within that hour the crawled documents had not changed at all. The delay observed for the re-crawl



[Crawl vs. Re-crawl, 25 US Universities (2 levels)]

[Crawl .vs ReCrawl 10 Frequently Changed Portals (2 levels)]

Figure 7: Crawling vs. re-crawling in WebRACE.

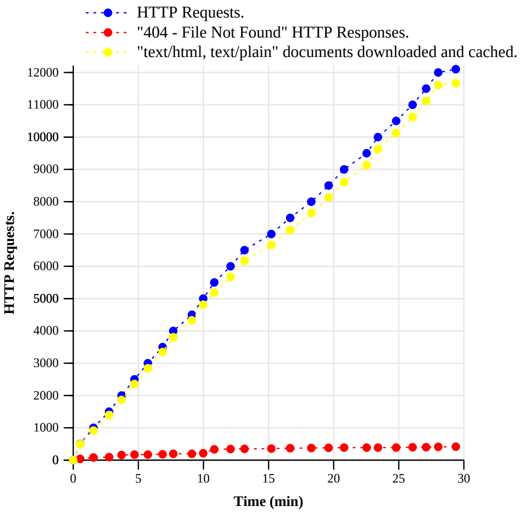
operation is attributed to the HTTP "if-modified-since" validation messages and the overhead of the Object Cache. As we can see from this diagram, the employment of the Meta-Info Store results to an almost three-fold improvement in the crawling performance. Moreover, it reduces substantially the network traffic and the Web-servers' load generated because of the crawl.

The diagram of Figure 7 (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. It should be noted, however, that within the first 100*msecs* of all crawl operations, crawling and re-crawling exhibit practically the same performance behavior. This is attributed to the fact that most of the crawled portals reply to our HTTP `GET` requests with "`301 (Moved Permanently)''` responses, and re-direct our crawler to other URL's. In these cases, the crawler terminates the connection and schedules immediately a new HTTP `GET` operation to fetch the requested documents from the re-directed address.

Finally, in Figure 8, we present measurements from a longer crawl that took 30*mins* to complete and produced 11669 documents. This crawl was conducted on our departmental Web server.

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



Crawling www.cs.ucy.ac.cy (10 levels, 45 URLfetchers)

Figure 8: Performance of a longer crawl.

to user-profiles are dropped from the cache.

Personalized annotation engines are not used in typical Search Engines [5], 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 9). Multiple *Filtering Processors* remove requests from the Coordinator and process them according to the *Unified Resource Descriptions (URD's)* of eRACE users contained in the request. Currently, the annotation engine implements a simple pattern-matching algorithm looking for weighted keywords that are included in the user-profiles.

6.1 URD's and ACI's

URD is an XML-encoded data structure that encapsulates source information, processing directives and urgency information for Web services monitored by eRACE. A typical URD request is shown in Table 4. The explanation of the URD scheme is beyond the scope of this paper.

URD's are stored in a single XML-encoded document, which is managed by a persistent DOM data manager (*PDOM*) [15]. The Annotation Engine fetches the necessary URD's from the *PDOM* data manager issuing XQL queries (eXtensible Query Language) to a GMD-IPSI XQL engine [15, 21]. The GMD-IPSI XQL engine is a Java-based storage and query application developed by Darmstadt GMD for handling large XML documents. This engine is based on two key mechanisms: a) a

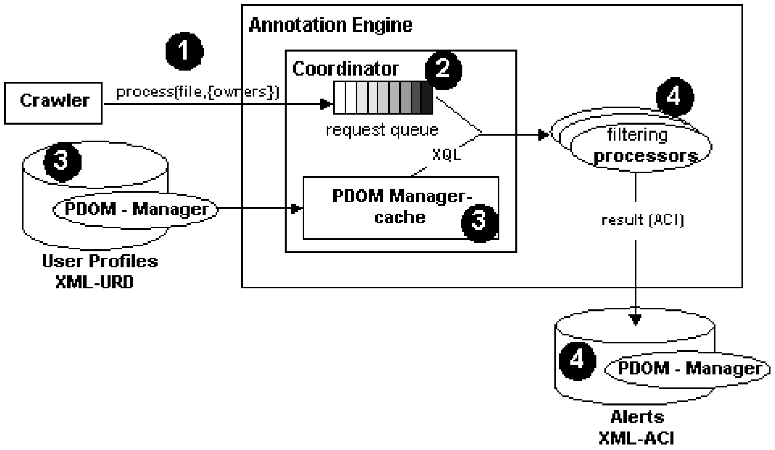


Figure 9: WebRACE Annotation Engine.

```
<urd>
   <uri timing= “600000” lastcheck = “97876750000” port= “80 >”
http://www.cs.ucy.ac.cy/default.html < /uri>
   <type protocol= “http” method= “pull” processtype= “filter”/ >
      <keywords>
           <keyword key= “ibm” weight= “1” / >
           <keyword key= “research” weight= “3” / >
           <keyword key= “java” weight= “4” / >
           <keyword key= “xmlp4j” weight= “5” / >
      < /keywords>
   <depth level= “4”/ >
   <urgency urgent= “1”/ >
< /urd>
```

Table 4: A typical URD.

```
public Element get(String id) {
   String query = ″//urd[@id= ´ ″ + id + ´ “]″;
   XQLResult r = XQL.execute(query, doc);
   Element urd = (Element) r.getItem(0);
}
```

Table 5: Retrieving a URD-XML node from PDOM.

persistent implementation of W3C-DOM Document objects [1]; b) a full implementation of the XQL query language. GMD-IPSI provides an efficient and reliable way to handle large XML documents through PDOM, which is a thread-safe and persistent XML-DOM implementation. PDOM supports main-memory caching of XML nodes, enabling fast searches in the DOM tree. A PDOM file is organized in pages, each containing 128 DOM nodes of variable length. When a PDOM node is accessed by a W3C-DOM method, its page is loaded into a main memory cache. The default cache size is 100 pages (12800 DOM nodes). Documents are parsed once and stored in Java serialized binary form on secondary storage. The generated document is accessible to DOM operations directly, without re-parsing. The XQL processor is used to query PDOM files. Table 5 illustrates the use of an XQL command to extract a URD-XML node out of PDOM.

The output of a filtering process in the Annotation Engine is encoded in XML and called an *ACI* (Annotated Cached Information) [28]; ACI's are stored in an XML-ACI PDOM database. ACI stands for Annotated Cached Information and is an extensible data structure that encapsulates information about the Web source that corresponds to the ACI, the potential user-recipient(s) of the "alert" that will be generated by eRACE's Content Distribution Agents according to the ACI, a pointer to the cached content, a description of the content (format, file size, extension), a classification of this content according to its urgency and/or expiration time, and a classification of the document's relevance with respect to the semantic interests of its potential recipient(s). The XML description of the ACI's is extendible and therefore we can easily include additional information in it without having to change the architecture of WebRACE. Table 6 gives an example of a typical ACI snippet. A more detailed description of the ACI scheme is beyond the scope of this paper.

6.2 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 (see Figure 10). 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

```
<aci owner = ‘‘csyiazt1’’ extension = ‘‘html’’ format= ‘‘html’’
        relevance= ‘‘18’’ updatetime= ‘‘97876950000 filesize= ‘‘2000’’>
   <uri>http://www.cs.ucy.ac.cy/default.html< /uri>
   <urgency urgent= ‘‘1’’/ >
   <docbase>969890.gzip< /docbase>
   <expired expir= ‘‘false’’ / >
   <summary>This is a part of the document with keywords 1)...< /summary>
< /aci>
```

Table 6: ACI snippet.

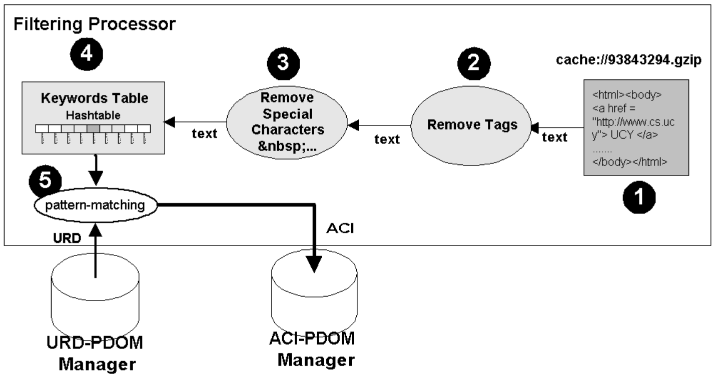


Figure 10: The Filtering Processor.

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 msecs to calculate the ACI for a 70KB Web page, with 3 potential recipients.

In our experiments, we have configured the SafeQueue size of the Annotation Engine to 1000 nodes, which is more than enough, since it is almost every time clear if the AE operates with 10 Filtering Processors and the Mini-crawler with 100 URL-fetchers. We have also observed that the number of pending requests in the AE SafeQueue has reached a peak of 55 pending requests at a particular run of our system.

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 [20, 14, 7, 6, 24], proxy services [4, 26], information dissemination systems [27, 2, 22] and Internet middleware [13, 9], 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. Further optimizations will be included in the near future, so as to prevent our crawler from overloading remote Web servers with too many concurrent requests. We also plan to investigate alternative queue designs and different crawling strategies (breadth-first versus depth-first) that have been reported to provide improved crawling efficiency [7]. Finally, we plan to investigate the employment of Distributed Data Structures [12] to further improve the scalability and performance of mission-critical components of WebRACE.

8 Acknowledgements

This work was supported partly by the grant PENEK-No 23/2000 from the Research Promotion Foundation of Cyprus, and by WinMob Technologies Ltd, Nicosia, Cyprus.

References

[1] Document Object Model (DOM) Level 1 Specification. W3C Recommendation 1, October 1998. http://www.w3.org/TR/REC-DOM-Level-1/.

[2] D. Aksoy, M. Altinel, R. Bose, U. Cetintemel, M.J. Franklin, J. Wang, and S.B. Zdonik. Research in Data Broadcast and Dissemination. In *Proceedings of the First International Con-*

ference on Advanced Multimedia Content Processing, AMCP '98, Lecture Notes in Computer Science, pages 194–207. Springer Verlag, 1999.

[3] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer protocol – HTTP/1.0. Technical report, W3C, May 1996. http://www.w3.org/Protocols/HTTP/1.0/spec.html.

[4] C. M. Bowman, P. B. Danzig, D. R. Hardy, U. Manber, and M. F. Schwartz. The Harvest Information Discovery and Access System. In *Proceedings of the Second International WWW Conference*, 1995.

[5] S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual (Web) Search Engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[6] S. Chakrabarti, M. van den Berg, and B. Dom. Focused Crawling: A New Approach to Topic-Specific Web Resource Discovery. In *8th World Wide Web Conference*, Toronto, May 1999.

[7] J. Cho, H. Garcia-Molina, and L. Page. Efficient crawling through URL ordering. In *Proceedings of the Seventh International WWW Conference*, pages 161–172, April 1998.

[8] M. Dikaiakos. FIGI: Using Mobile Agent Technology to Collect Financial Information on Internet. In *Workshop on Data Mining in Economics, Marketing and Finance. Machine Learning and Applications. Advanced Course on Artificial Intelligence 1999 (ACAI '99).* European Coordinating Committee on Artificial Intelligence and Hellenic Artificial Intelligence Society, July 1999.

[9] P. Farjami, C. Gorg, and F. Bell. Advanced Service Provisioning Based on Mobile Agents. *Computer Communications*, (23):754–760, 2000.

[10] J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer protocol – HTTP/1.1. Technical report, W3C, June 1999. http://www.w3.org/Protocols/rfc2616/rfc2616.html.

[11] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[12] S. Gribble, E. Brewer, J. Hellerstein, and D. Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*, 2000.

[13] S. Gribble, M. Welsh, R. von Behren, E. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R.H. Katz, Z.M. Mao amd S. Ross, and B. Zhao. The Ninja Architecture for Robust Internet-Scale Systems and Services. *To appear in a Special Issue of Computer Networks on Pervasive Computing*, 2001.

[14] A. Heydon and M. Najork. Mercator: A Scalable, Extensible Web Crawler. *World Wide Web*, 2(4):219–229, December 1999.

[15] G. Huck, I. Macherius, and P. Fankhauser. PDOM: Lightweight Persistency Support for the Document Object Model. In *Proceedings of the 1999 OOPSLA Workshop Java and Databases: Persistence Options. Held on the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '99).* ACM, SIGPLAN, November 1999.

[16] Tympani Development Inc. NetAttache Pro. http://www.tympani.com/products/NAPro.html, 2000.

[17] S. Lempinen. *Jtidy.* http://lempinen.net/sami/jtidy.

[18] Steve Meloan. The Java HotSpotTM Performance Engine: An In-Depth Look. Technical report, Sun Microsystems, June 1999. http://developer.java.sun.com/developer/technicalArticles/Networking/HotSpot/.

[19] Sun Microsystems. The Java HotSpot TM Server VM. http://java.sun.com/products/hotspot/, 1999.

[20] R. Miller and K. Bharat. SPHINX: A Framework for Creating Personal, Site-specific Web Crawlers. In *Proceedings of the Seventh International WWW Conference*, pages 161–172, April 1998.

[21] GMD-IPSI XQL Engine. http://xml.darmstadt.gmd.de/xql/.

[22] S.H. Phatak, V. Esakki, B.R. Badrinath, and L. Iftode. Web&: An Architecture for Non-Interactive Web. Technical Report DCS-TR-405, Department of Computer Science, Rutgers University, December 1999.

[23] D. Raggett. *Clean up your Web pages with HTML TIDY.* http://www.w3.org/People/Raggett/tidy/.

[24] S. Raghavan and H. Garcia-Molina. Crawling the Hidden Web. In *VLDB 2001: 27th International Conference on Very Large Data Bases*, September 2001. To appear.

[25] VMGEAR. OptimizeIt!: The Java Ultimate Performance Profiler. http://www.vmgear.com/.

[26] D. Wessels and K. Claffy. Evolution of the NLANR Cache Hierarchy: Global Configuration Challenges. Technical report, NLANR, October 1996. http://www.nlanr.net/Papers/Cache96/.

[27] T. W. Yan and H. Garcia-Molina. SIFT - A Tool for Wide-Area Information Dissemination. In *Proceedings of the 1995 USENIX Technical Conference*, pages 177–186, 1995.

[28] D. Zeinalipour-Yazti. eRACE: an eXtensible Retrieval, Annotation and Caching Engine, June 2000. B.Sc. Thesis. In Greek.