A Distributed Middleware Infrastructure for Personalized Services[*]

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

Demetris Zeinalipour-Yazti
Dept. of Computer Science
University of California
Riverside, CA, USA
csyiazti@cs.ucr.edu

Abstract

In this paper we present an overview of eRACE, a modular and distributed intermediary infrastructure that collects information from heterogeneous Internet sources according to registered profiles or end-user requests. Collected information is stored for filtering, transformation, aggregation, and subsequent personalized or wide-area dissemination on the wireline or wireless Internet. We study the architecture and implementation of the main module of eRACE, an HTTP proxy named *WebRACE*. WebRACE consists of a high-performance, distributed and multithreaded Web crawler, a multithreaded filtering processor and an object cache. We discuss the implementation of WebRACE in Java, describe a number of performance optimizations, and present its performance assessment.

1 Introduction

The rapid expansion of the Web, and the developments in mobile-device and wireless-Internet technologies have resulted to a large heterogeneity of client devices currently used for accessing Internet services. Furthermore, they have raised the capacity mismatch between clients and Internet servers. To cope with these trends, software infrastructures for

[*] Work supported in part by the Research Promotion Foundation of Cyprus under grant PENEK 23/2000 and by the European Union under the ANWIRE project (contract IST-2001-38835).

Internet services have to: (i) Support seamless access from a variety devices; (ii) Customize content according to the requirements and limitations of different terminal devices. (iii) Support both synchronous (on-demand) and asynchronous modes of interaction with users, thus coping with frequent disconnections of wireless access and user mobility. (iv) Optimize the amount of useful content that reaches users through client devices with limited resources and restricted interfaces, by enabling service personalization, localization and filtering of information. (v) Guarantee high availability and robustness, as well as incremental performance and capacity scalability with an expanding user base.

Possible approaches for coping with these requirements are the so-called end-to-end solutions, where origin servers adapt their content on-the-fly, taking into account consolidated user profiles [43] or the terminal device and network connection involved in a user session [35]. For an end-to-end approach to work properly, however, adaptation software has to be inserted at each origin server. Consequently, software updates have to be propagated to all origin servers whenever new terminal devices and content-encoding protocols emerge. Moreover, on-the-fly adaptation of content can be very time-consuming, leading to a deterioration of user experience [21].

An alternative approach for providing personalization, content customization, ubiquity and mobility, is the employment of proxies with a functionality significantly extended over what is found in traditional proxies for the wireline or wireless Web [21]. Given the high-performance requirements for high throughput, 24x7 availability, and performance scalability of next-generation Internet services [11, 40], however, centralized proxies are expected to face performance problems as well, thus necessitating the distribution of their computation, storage and complexity into the networking infrastructure [21]. In such a case, a number of distributed, programmable and possibly mobile *intermediary servers* would be deployed throughout the network. These servers would mediate between primary information sources and various client systems, providing performance scalability, better sharing of resources, higher cost efficiency, and a streamlining of new service provision [10].

The focus of our work is on the development of eRACE, an *intermediary* infrastructure with enhanced functionality and distributed architecture, to support the development and deployment of personalized services and the provision of ubiquitous access thereof. In this paper we present the design principles and architecture of eRACE. Furthermore, we describe the design, implementation and performance assessment of *WebRACE*, which is an eRACE-proxy dealing with collecting, filtering and caching content from the World-Wide Web, according to personal and service profiles. WebRACE consists of a multithreaded crawler, a multithreaded filtering engine, and an object cache. The remaining of this paper is organized as follows: Section 2 presents an overview of the eRACE architecture. Section 3 describes the main challenges behind WebRACE design and implementation. Sections 4, 5 and 6 describe the main components of WebRACE: the Mini-crawler, the Object Cache and the Annotation Engine. Section 7 presents our experimentation and performance assessment of WebRACE. Furthermore, it presents the performance enhancements that we achieve by caching to cache the crawling state in the Object Cache, and by distributing the crawler to a network of workstations. Section 8 presents an overview of recent related work. We provide our conclusions in Section 9.

2 The Architecture of eRACE

2.1 Overview and Goals

The *extensible Retrieval, Annotation and Caching Engine (eRACE)* is a middleware infrastructure designed to support the development and deployment of *intermediaries* on Internet. Intermediaries are "software programs or agents that meaningfully transform information as it flows from one computer to another" [8, 38], and represent a useful abstraction for describing and developing personalized proxies, mobile services, etc.

eRACE is a modular, configurable, and distributed proxy infrastructure that collects information from heterogeneous Internet sources and protocols according to *eRACE profiles* registered within the infrastructure, and end-user requests. Collected information is stored in the software cache for further processing, personalized dissemination to subscribed users, and wide-area dissemination on the wireline or wireless Internet.

eRACE supports personalization by enabling the registration, maintenance and management of personal profiles representing the interests of individual users. Furthermore, its structure allows the easy customization of service provision according to parameters, such as information-access modes (pull or push), client-proxy communication (wireline or wireless; email, HTTP, WAP), and client-device capabilities (PC, PDA, mobile phone, thin clients). Ubiquitous service-provision is supported by eRACE thanks to the decoupling of information retrieval, storage and filtering, from content publishing and distribution. The eRACE infrastructure can also easily incorporate mechanisms for providing subscribed users with differentiated service-levels at the middleware level. Finally, the design of eRACE is tuned for providing performance scalability, which is an important consideration given the expanding numbers of WWW users, the huge increase of information sources available on the Web, and the need to provide robust services.

Key design and implementation decisions made to accomplish these goals are described below:

1. The information architecture of eRACE is defined in terms of metadata that represent user account and connection information, user and service profiles, state information of eRACE modules and information exchanges taking place between them. Specifications are defined as XML Document Type Definitions (DTDs) [51]. We chose XML because it is simple, self-descriptive, and extensible. Hence, we can easily extend our descriptions to incorporate new services, terminal devices, and QoS policies. Furthermore, we can re-use existing modules and API's that process XML data. Central to this set of meta-data is the "eRACE profile" and the "eRACE annotation." The *eRACE profile* is a concise XML description of the operations and transformations that eRACE modules are expected to perform upon heterogeneous Internet sources: information gathering, filtering and caching of retrieved content, transcoding, dissemination, etc. The eRACE-profile DTD is general and expressive so as to represent: (i) personal interests of subscribers, thus supporting the deployment of personalized services over the Web; (ii) generic services that can be deployed on wide-area networks, such as notification systems, portals, mobile services, etc. The results of profile-driven, filtering operations performed upon gathered content are encoded in XML and named *eRACE Annotations* or *Annotation Cache Information* (ACI's).

2. Data sharing between modules of the eRACE infrastructure is done through messages that transport XML-encoded information and events. Therefore, we can easily decouple and isolate modules from each other and distribute them physically across machine boundaries at configuration or run-time. Furthermore, modules with stringent performance requirements are multithreaded and employ distributed data-structures [29], to

allow the exploitation of parallel execution on shared-memory multiprocessor systems and networks of workstations.

3. eRACE translates user requests and eRACE profiles into “eRACE requests,” encoded in XML and tagged with QoS information. These requests are scheduled for execution by an eRACE scheduler, which can implement different scheduling policies based on QoS tags. The explicit maintenance of XML-encoded information regarding pending requests and content scheduled for dissemination, makes it easy to keep track of the system’s run-time behavior, to compare alternative scheduling algorithms, to implement load-balancing techniques for sustaining high-availability during high loads, and to apply QoS policies with different service levels.

4. eRACE is implemented with Java [28]. 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 modules. Java class libraries provide support for key features of eRACE: platform independence, multithreading, network programming, high-level programming of distributed applications, string processing, code mobility, compression, etc. Our choice of Java, however, came with a certain risk-factor that arose from known performance problems of this platform and its run-time environment. Performance and robustness are issues of critical importance for systems like eRACE, which must perform as a server, run continuously and sustain high-loads at short periods of time.

5. Support for mobility and disconnected operations of Proxy and Content-Distribution agents will be provided by Mitsubishi’s Concordia Mobile Agent platform [36, 37]. For that matter, we have conducted a number of studies to assess the performance and robustness of this platform, with encouraging results [23, 48]. Furthermore, we implemented two earlier prototypes prototype of eRACE [22, 52] with Concordia; mobile agents were used to implement the communication protocol between users and eRACE servers. To this end, the front-end interface of eRACE was implemented as a Java applet with an embedded transporter able to launch and receive Concordia Agents. “Agent proxies” were implemented as stationary Concordia Agents able launching mobile agents to access and combine information sources over the network.

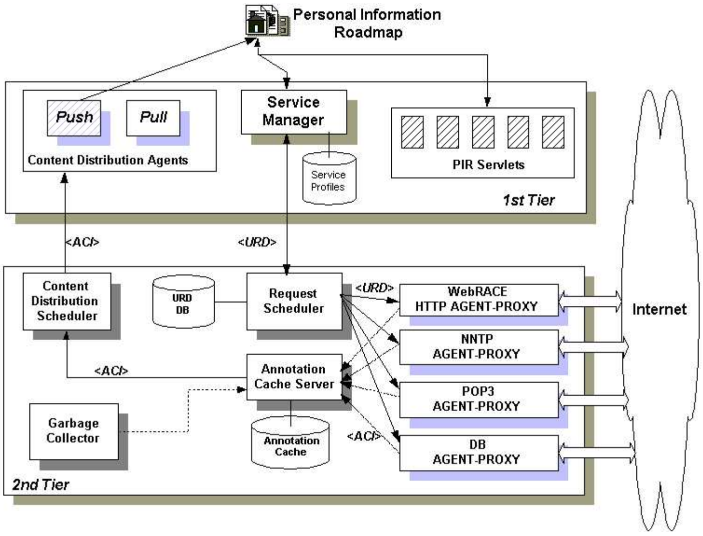


Figure 1: eRACE System Architecture.

2.2 System Architecture

2.2.1 First Tier

eRACE is organized as a two-tier architecture (see Figure 1). The first tier includes modules that manage services provided to users: the Service Manager, Content-Distribution Agents, and Personal Information Roadmap (PIR) Servlets.

The *Service Manager* is comprised of modules for handling user connection (authentication, login) and profile management. Each time a user connects to eRACE, the Service Manager notifies other modules so that content starts "flowing" to the user. Furthermore, the Service Manager maintains and manages user and service profiles that are defined and stored as XML data. A profile is a set of long-term, continuously evaluated queries [50]; in eRACE, these queries can be typical queries to Web databases, HTTP requests for World-Wide Web resources, access to general-purpose Search Engines or Subject Cataloging Sites, subscription to Usenet News, etc. Each profile is annotated with a number of *data* and *control* parameters. Data parameters are query arguments, e.g., a stock symbol of interest. Control parameters determine the frequency of query execution, the expected amount of information gathered from queries (e.g., summary vs. full results), the priority of notification for a given query, etc. Profiles are managed by the Service Manager through the Java API of PDOM [34], which is a thread-safe, persistent Document Object Model data manager.

The Service Manager translates these profiles into eRACE requests that are forwarded for execution to the second tier of eRACE (see Figure 1).

Content-Distribution Agents retrieve content pertinent to user and service profiles, and maintained in the caches of eRACE's Agent Proxies (with meta-information kept in the Annotation Cache). Content is aggregated by the CDA's, which decide when and how to disseminate it to end-users, following possible terminal- or connection-specific transformations (adaptation, transcoding, etc.). Additional optimizations, such as caching content within the Content-Distribution Agents "near" the PIR Servlets, will be addressed in future work.

eRACE provides users with seamless access to their content through the *Personal Information Roadmap* (PIR). This is a customized user-interface that implements a simple e-mail based information provision paradigm, seeking to cope with problems of network disorientation and information overloading. The PIR provides a user with a personalized and unified view of her personal information space across different devices. The PIR is implemented as a set of Java Servlets, which transcode it to a format appropriate for the users' terminal connections (HTML for connections over HTTP and WML for connections over WAP). The PIR can be used simultaneously with other tools employed to access information on Internet (browsers, e-mailers, etc).

2.2.2 Second Tier

The second tier of eRACE consists of a number of protocol-specific *agent-proxies* like WebRACE, mailRACE, newsRACE and dbRACE that retrieve and cache information from the WWW, POP3 email-accounts, USENET NNTP-news, and Web-database queries respectively (see Figure 1).

Agent-proxies are driven by a *Request Scheduler*, which scans continuously a database of *Unified Resource Descriptions* (URD-DB in Figure 1) and schedules URD's for execution to the corresponding agent-proxy. A *Unified Resource Description* (URD) is an XML-encoded data structure, which is part of an "eRACE request," and describes source information, processing directives and urgency information for Internet sources monitored by eRACE. In essence, each URD represents a request to retrieve and process content from a particular Internet source on behalf of a particular user or service. A typical URD request is shown

```
<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 1: A typical URD instance.

in Table 1.

The URD database (URD-DB) is populated by the Service Manager of eRACE. URD-DB is a single XML-encoded document managed by *PDOM* [34]. XML documents are parsed by PDOM and stored in Java serialized binary form on secondary storage, organized in pages, each containing 128 DOM nodes of variable length. The parsed document is accessible to DOM operations directly, without re-parsing. PDOM nodes accessed by a DOM operation are loaded into a main memory cache. PDOM supports main-memory caching of XML nodes, enabling fast searches in the DOM tree.

Access to URD-DB's contents is provided through the data manager of *PDOM*, which issues XQL queries (eXtensible Query Language) to a GMD-IPSI XQL engine [34, 42]. This engine is a Java-based storage and query application, which handles large XML documents and incorporates two key mechanisms: a persistent implementation of W3C-DOM Document objects [2], and a full implementation of the XQL query language.

The content produced by a URD execution is stored in the software cache of the corresponding agent-proxy and filtered by the Annotation Engine of eRACE according to processing directives defined in that URD. Meta-information produced by the filtering process is encoded as an XML data-structure named *Annotation Cache Information* (ACI) and

```
<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 2: ACI snippet.

cached separately. ACI's are used by *Content-Distribution Agents* for information dissemination to end-users. ACI 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. ACI's are stored in an XML-ACI PDOM database. An example of a typical ACI snippet is given in Table 2.

Agent-proxies implement a number of optimizations such as coalescing different URD requests that target the same information source. Furthermore, agent-proxies implement expiration policies that differ from the expiration policies of information sources on Internet. eRACE maintains and manages multiple versions of the content published on some information source every time this content is of interest to some eRACE profile, and for as long as the resulting information has not been retrieved by interested end-users. This approach makes it necessary to manage obsolete information stored in eRACE caches explicitly. This task is carried out by a Garbage Collector module (see Figure 1).

3 WebRACE Design and Implementation Challenges

WebRACE is the agent-proxy that deals with information sources on the WWW and accessible through the HTTP protocols (HTTP/1.0, HTTP/1.1). Other proxies have the same general architecture with WebRACE, differing only in the implementation of their protocol-specific proxy engines.

WebRACE is comprised of three basic components: the *Mini-crawler*, the *Object Cache*, and the *Annotation Engine*. These components operate independently and asynchronously (see Figure 2). They can be distributed to different computing nodes, execute in different Java heap spaces, and communicate through permanent socket links. Through these sockets, 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 [12], 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 2). These requests change dynamically with shifting eRACE-user interests, updates in the base of registered users, changes in the set of monitored resources, etc.

A second challenge is to design 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 [39], 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

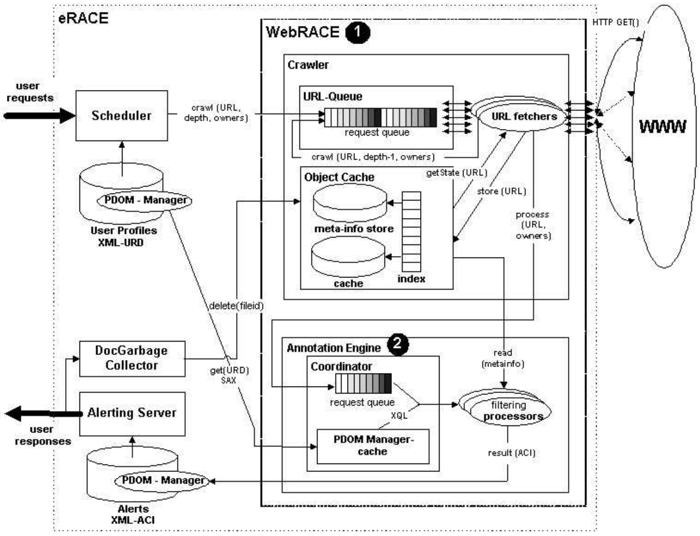


Figure 2: WebRACE System Architecture.

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 [12], Mercator [33], and SPHINX [39].

Finally, WebRACE is implemented entirely in Java v.1.3. Extensive performance and memory debugging with the OptimizeIt profiler [49], however, identified a number of performance problems arising because of Java core classes (excessive allocation of new objects causing heap-space overflows and performance degradation. Consequently, we developed 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 information on implementation details can be found in [54, 53].

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 [33]: 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 [39, 33], 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 configuration files and can be adapted 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* that uses HTTP to download documents from the Web. The URLFetcher contains also a *URL extractor and normalizer* that 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.

4.1 The URLQueue

The *URLQueue* is an implementation of the *SafeQueue* data structure that we designed and implemented to achieve the efficient and robust operation of WebRACE and to overcome problems of the `java.util.LinkList` component of Java [28]. SafeQueue is a circular array of *QueueNode* objects with its own memory-management mechanism, enabling the re-use of objects and minimizes garbage-collection overhead. Moreover, SafeQueue provides support for persistence, overflow control, disk caching, multi-threaded access, and fast indexing to avoid the insertion of duplicate QueueNode entries [54, 53].

URLQueue is a SafeQueue subclass comprised of $URLQueueNode$'s, i.e., Java objects that represent requests coming from the Request Scheduler of eRACE. During the server's initialization. The length of the URLQueue is determined during WebRACE initialization from its configuration files. At initialization time, WebRACE allocates the heap-space required to store all the nodes of the queue. This approach is chosen instead of allocating Queue Nodes on demand for memory efficiency and performance. In our experiments, we 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.

4.2 The URLFetcher

The *URLFetcher* is a multithreaded WebRACE that fetches documents from the Web using the HTTP/1.0 and HTTP/1.1 protocols. URLFetcher threads retrieve 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.

In addition to handling HTTP connections, the URLFetcher processes downloaded documents. To this end, it invokes 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. As shown in Figure 3, the URLExtractor and normalizer

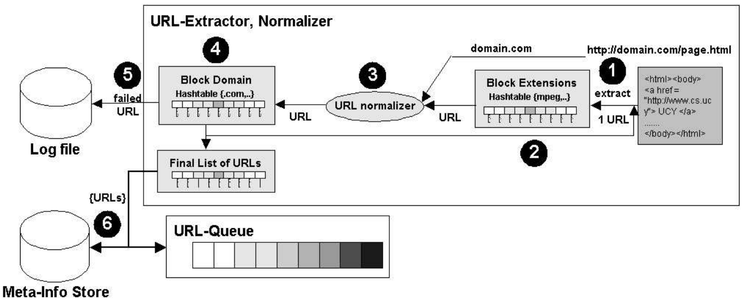


Figure 3: URL Extractor Architecture

works as a 6-step pipe within the URLFetcher.

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.

The Object Cache is comprised of an *Index*, a *Meta-info Store* and an *Object Store* (see Figure 2). 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.

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: (i) The URL address of the corresponding document; (ii) The IP address of its origin Web server; (iii) The document size in KiloBytes; (iv) The Last-Modified field returned by the HTTP protocol during download; (v) 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.

```
< 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.

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.

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

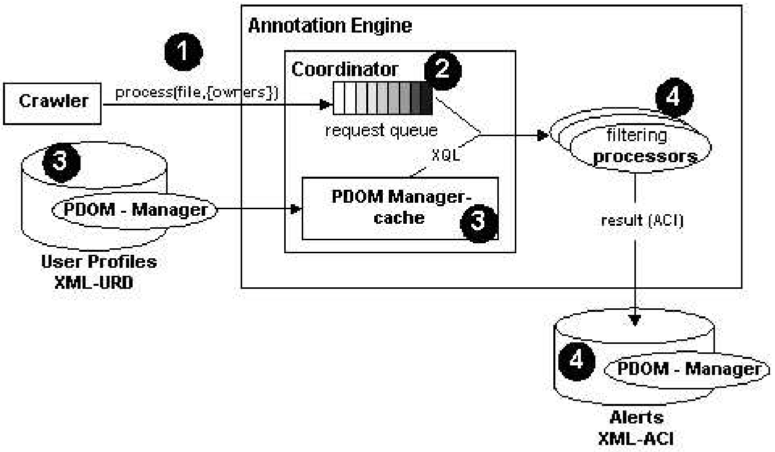


Figure 4: WebRACE Annotation Engine.

of the Annotation Engine is stored as eRACE annotations (ACI's) linked to the cached content. Pages that are not relevant to any user-profile are dropped from the cache.

Personalized annotation engines are not used in typical Search Engines [12], 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 downloaded pages "on the fly." Therefore, each time the Annotation Engine receives a `‘‘process(file,{users})’’` request through established socket connections with the Mini-crawler, it inserts the request in the *Coordinator*, which is a SafeQueue data structure (see Figure 4). 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 well-known string-matching algorithm looking for weighted keywords that are included in the user-profiles [50].

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 5). 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

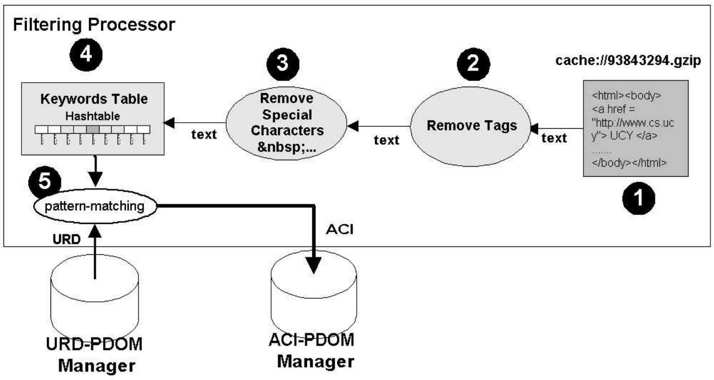


Figure 5: The Filtering Processor.

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.

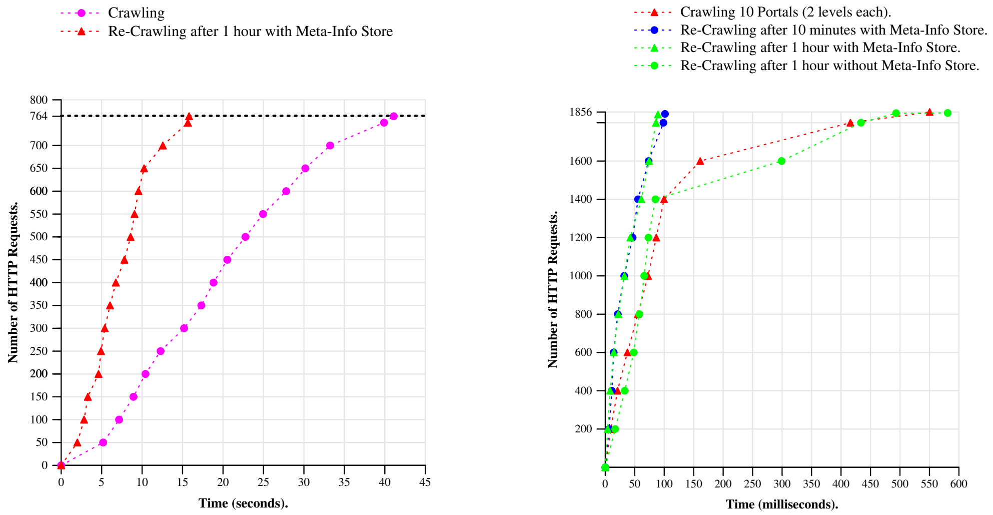
In our experiments, we have configured the SafeQueue size of the Annotation Engine to 1000 nodes, which proved to be adequately large in experiments conducted with 10 Filtering-processor threads and 100 URLFetcher threads.

7 Experiments and Performance Assessment

To evaluate the performance of WebRACE we ran a number of tests and extracted measurements of performance-critical components. In particular, we investigated the benefits of caching the crawling state, the overall performance of the crawler, and the effects that multithreading has on URLFetcher performance. For our experiments, we crawled three classes of Web sites: the first class includes servers that provide content which does not change very frequently (remote University sites in the U.S.); the second class consists of popular news-sites, search-engine sites and portals (cnn.com, yahoo.com, msn.com, etc.); the third class consists of Web servers residing in our departmental network.

7.1 Overall Performance and Benefits from Caching Crawling State

To assess the performance improvement achieved caching crawling state in the Meta-info Store, we conducted experiments with the first two classes of Web sites. For these experiments we configured WebRACE to use 150 concurrent URLFetchers and ran it on our



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

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

Figure 6: Crawling vs. re-crawling in WebRACE.

dual-processor Sun Enterprise E250 Server, with the Annotation Processor running concurrently with 10 threads on a Sparc 5.

The diagram of Figure 6 (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 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 6 (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: from a throughput of 3.37 requests/sec of the first crawl, we achieve throughput of 18.31 and 20.77 requests/sec for the two subsequent re-crawls, respectively.

It should be noted that within the first 100 msecs of all these crawling experiments, 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

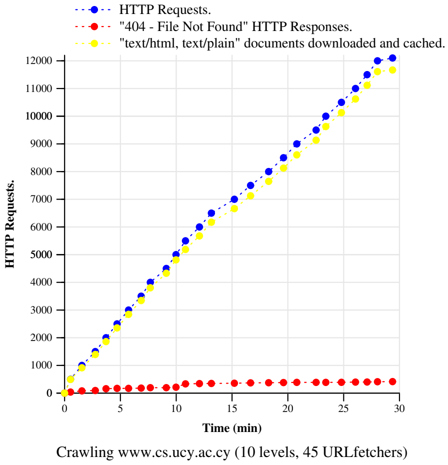


Figure 7: Performance of a longer crawl.

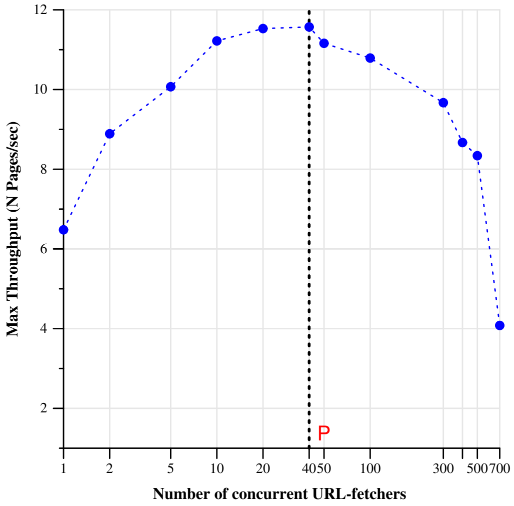
“`301 (Moved Permanently)''` responses, and re-direct our crawler to other URL’s. In these cases, the crawler terminates the connection and schedules immediately a new HTTP `GET` operation to fetch the requested documents from the re-directed address.

Finally, in Figure 7, we present measurements from a longer crawl that took 29.38 mins to complete and produced 11669 documents. This crawl was conducted on our (slow) departmental Web servers.

7.2 URLFetcher Performance

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 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 8, the throughput increases with the number of concurrent URLFetchers, until a peak P is reached. After that point, throughput drops substantially, because of the synchronization overhead incurred by the large number of concurrent threads. 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



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

Figure 8: URLFetcher throughput degradation.

optimal crawling throughput.

7.3 Distributed Mini-Crawler Execution

In order to speed-up the performance of WebRACE, we sought to parallelize the Mini-Crawler by running multiple multithreaded URLFetcher's on a network of workstations, keeping the number of threads at each workstation optimal. To this end, we employed the Java-based *Distributed Data Structure* component, which was developed at UC/Berkeley [29]. The DDS component distributes and replicates the contents of a data-structure across a cluster or network of workstations, providing the programmer with a conventional, single-site, in-memory interface to the data structure [29]. Coherence between distributed replicas is established through an implementation of a two-phase commit protocol, which is transparent to the programmer of services on top of DDS.

When using the DDS layer and API, we kept the core of WebRACE intact and added-on a module that could handle the distribution of the Mini-Crawler. To this end, we ran multiple URLFetchers on different machines and used the Distributed Hashtable implementation of DDS [29] to index the documents gathered by the URLFetchers, allowing the crawlers to know the content of the Meta-info store. The total time spent to incorporate the DDS in WebRACE and make it distributed was less that 3 hours.

The topology of the distributed Mini-Crawler is presented in Figure 9. We employed

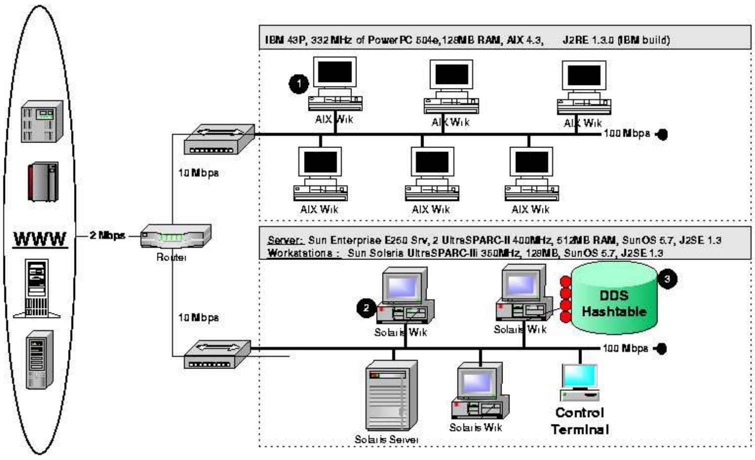
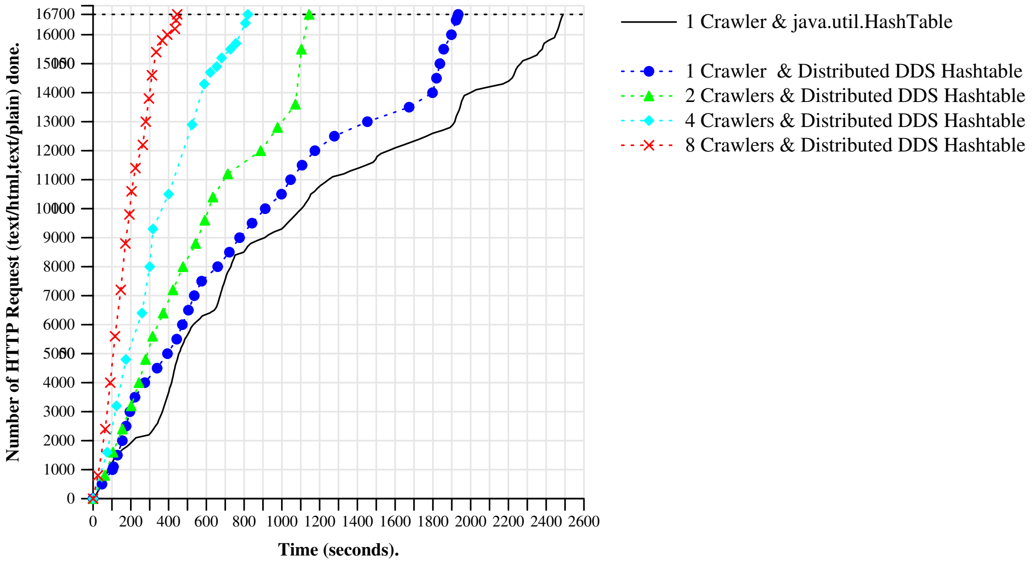

IBM 43P, 332 MHz of PowerPC 604e,128MB RAM, AIX 4.3, J2RE 1.3.0 (IBM build)
1
AIX Wik
AIX Wik
AIX Wik
100 Mbps
AIX Wik
AIX Wik
AIX Wik
10 Mbps
WWW
2 Mbps
Router
Server: Sun Enterprise E250 Srv, 2 UltraSPARC-II 400MHz, 512MB RAM, SunOS 5.7, J2SE 1.3
Workstations : Sun Solaris UltraSPARC-IIi 350MHz, 128MB, SunOS 5.7, J2SE 1.3
2
Solaris Wik
Solaris Wik
DDS Hashtable
3
10 Mbps
100 Mbps
Solaris Server
Solaris Wik
Control Terminal

Figure 9: WebRACE Distributed Crawler Network Topology.

eight crawling nodes, one DDS indexing node, and one control terminal for monitoring and control. The crawling nodes are IBM 43P workstations running AIX, SPARC workstations running Solaris, and a dual processor Sun Enterprise E250 server, all connected on our 10/100 Mbps Local Area Network.

To assess the performance improvement provided by the distribution of WebRACE, we conducted an experiment crawling our first class of servers (University sites) in a depth of 3 levels. To this end, we configured each crawling node to use only 45 concurrent URLFetchers. Measurements are presented in Figure 10, which shows that the performance of the distributed WebRACE scales linearly with the number of crawling nodes. It is interesting to note that for 45 concurrent URLFetcher threads, the performance of the distributed WebRACE with one crawling node is better than that of the "standalone" WebRACE. This is attributed to the poor performance of the synchronized `java.util.HashTable` used in the latter case. Finally, different crawling nodes display a different crawling throughput: on the average, the E250 dual processor server was three times faster than the SPARC-stations, and almost ten times faster than the AIX machines.



[Crawling with multiple hosts 25 US Universities (3 levels)]

Figure 10: WebRACE Distributed Crawler Performance.

8 Related Work

8.1 Intermediaries

Research and development on intermediary software for Internet has produced a variety of systems, ranging from proxy servers [18], subscription-based notification and information dissemination services [50, 25, 5, 44, 22], tools that support mobile Web access [27], and more generic, programmable infrastructures that can be used to build ubiquitous services [6, 10, 30, 26]. These systems can be analyzed in the context of the architectural and functional specifications for "Open Pluggable Edge Services" (OPES), which are proposed by the IETF's OPES working group for describing intermediary services in open, distributed settings [1, 3]. Intermediary software can be characterized as "middleware," since it provides a "reusable and expandable set of services and functions, commonly needed by many applications to function well in a networked environment" [4]. Here, we describe some characteristic examples of intermediary infrastructures. A more detailed survey can be found in [21].

A programmable framework for building various intermediary services is the *Web Browser Intelligence* or *WeB Intermediaries* (WBI) by IBM Almaden [7, 6, 38]. WBI is a programmable proxy server designed for the development and deployment of intermediary applications. The design of WBI is based on the notion of "information streams" that convey data from information providers (e.g., a Web server) to information consumers (e.g., a Web browser), and can be classified into unidirectional and bidirectional messages or transaction streams [8]. WBI has been used to develop a number of applications such as a manager-repository for *cookies* and a Web-browsing service for mobile devices [6]. WBI plugins can be installed both on a client machine and on any other networked machine, possibly near an origin server. Multiple client-side and server-side WBI intermediaries can cooperate to establish one WBI service.

Building mobile services from proxy components is the main goal of the *iMobile* project of AT&T Research [46, 15]. The iMobile proxy maintains user and device profiles, accesses and processes Internet resources on behalf of the user, keeps track of user interaction and performs content transformations according to device and user profiles. The architecture of iMobile is established upon the infrastructure of *iProxy*, a programmable proxy server designed to host agents and personalized services developed in Java [47]. iMobile consists of three main abstractions: *devlets*, *infolets* and *applets*. A devlet is an agent-abstraction for supporting the provision of iMobile services to different types of mobile devices connected through various access networks. A devlet-instance communicates with a device-specific "driver" that is either co-located in the iProxy server or resides at a remote mobile support station. The infolet abstraction provides a common way of expressing the interaction between the iMobile server and various information sources or information spaces (in iMobile terminology) at a level higher than the HTTP protocol and the URI specification. Different sources export different interfaces to the outside world: JDBC and ODBC for corporate databases, the X10 protocol for home networks, IMAP for email servers, etc. Finally, an iMobile applet is a module that processes and aggregates content retrieved by different sources and relays results to various destination devices. At the core of an iMobile server resides the "let engine," which registers all devlets, infolets and applets, receives commands from devlets, forwards them to the right infolet or applet, transcodes the result to an appropriate terminal-device format and forwards it to the terminal device via the proper devlet.

A broader approach that seeks to develop a robust infrastructure and programming platform for Internet-scale systems and services in Java comes from the Ninja project at UC/Berkeley [30]. The architecture of Ninja consists of *bases*, *active proxies*, *units* and *paths.* Bases are scalable platforms designed to host Internet services. They consist of a programming model and I/O substrate designed to provide high-concurrency, robustness, and transparent distribution of data to cluster-nodes [29]. Moreover, they include a cluster-based execution environment (*vSpace*) that provides facilities for service component replication, load-balancing and fault-tolerance [30]. The programming model of Ninja consists of four *design patterns* that service programmers can use to compose different stages of a single service: wrap, pipeline, combine and replicate [30]. Active proxies are fine-grain intermediaries providing transformational support between Ninja services and terminal devices. Active proxies perform data distillation, protocol adaptation, caching, encryption, etc. Examples of active proxies include wireless base-stations, network gateways, firewalls, and caching proxies. In Ninja terminology, a path is a flow of typed data through multiple proxies across a wide-area network; each proxy performs transformational operations to adapt the data into a form acceptable by the next service or device along the path. Similarly to WBI plugins, Ninja-paths can be established dynamically. Finally, units are abstractions for the client devices attached to the Ninja infrastructure, which range from PC's and laptops to mobile devices, sensors and actuators.

8.2 Crawlers

Crawlers have been a topic of intensive research in a number of contexts [12, 39, 17, 41, 45, 13, 16]. Here, we provide a brief overview of crawlers that bare some resemblance with WebRACE in terms of their design or scope. In particular, we focus on the following four classes of crawlers: *i) parallel, ii) personal, iii) focused* and *iv) peer-to-peer.*

Parallel crawlers employ multiple, concurrent crawling processes to minimize the time of large crawls. Typically, large crawlers are used by popular search engines, Web caches, and other large information retrieval systems. Despite the wide deployment of parallel crawlers, very few published studies have investigated crawling strategies [20] or evaluated crawler performance [16, 41]. Cho and Molina [16] explore the trade-offs of crawler parallelization and identify a number of parameters that are important in the design of a parallel crawler. According to this work, the main challenge confronting parallel crawlers is how to communicate internal state (i.e. indices, list of pending and downloaded URLs) among the several crawling processes. This challenge arises only for parallel crawlers that work on overlapping sets of URLs; other parallel crawlers restrict the visits of the different processes to separate, pre-determined sets of URLs [1]. Ideally, a crawling system tries to minimize the *overlap* among the concurrent processes, which is achieved by communicating internal state, while at the same time minimize the communication among the various processes distributed across several hosts.

ii) Personal crawlers, such as WebSPHINX [39] and the *Competitive Intelligence (CI) Spider* [14], allow users to "dispatch" simple crawlers on the Web without the need of acquiring access to dedicated crawling infrastructures. WebSPHINX includes a crawl visualization module, which allows its user to visually inspect accessed pages and broken links. Similarly to WebRACE, WebSPHINX supports exhaustive traversal techniques (breadth-first or depth-first traversal), multithreading, and JAVA. Nevertheless, it is designed for single-user usage and therefore does not support runtime scheduling of several crawling requests. The CI Spider collects user specified web pages and performs linguistic analysis and clustering of results. Such results are expected to help users or corporate managers to obtain critical information about some competitive environment (e.g. other companies) and to make informed decisions about important issues (such as investment, marketing or planning). CI Spider is designed to run on a client machine and therefore it is questionnable whether it provides the necessary performance scalability in order to extend its coverage to very large sub-domains of the Web.

iii) Focused crawlers: The inability of traditional personal and parallel crawlers to contextually prioritize the crawling sequence of URLs led to the development of *focused crawlers* [13], that is crawlers seeking and acquiring pages on a specific set of *topics. Topics* are usually provided offline as collections of contextually related documents, such as the documents organized in Web taxonomies (Yahoo! or Dmoz Open Directory). These collections are subsequently utilized for training some given classifier (e.g. a "Naive" Bayes Classifier [32]). The classifier will then be able to make decisions on whether the crawler should expand on a given URL or whether some URL should be discarded. The selection of the right training set and classifier are critical in the success of a focused crawler, as short-term crawling path decisions may supersede crawling paths that will ultimately lead

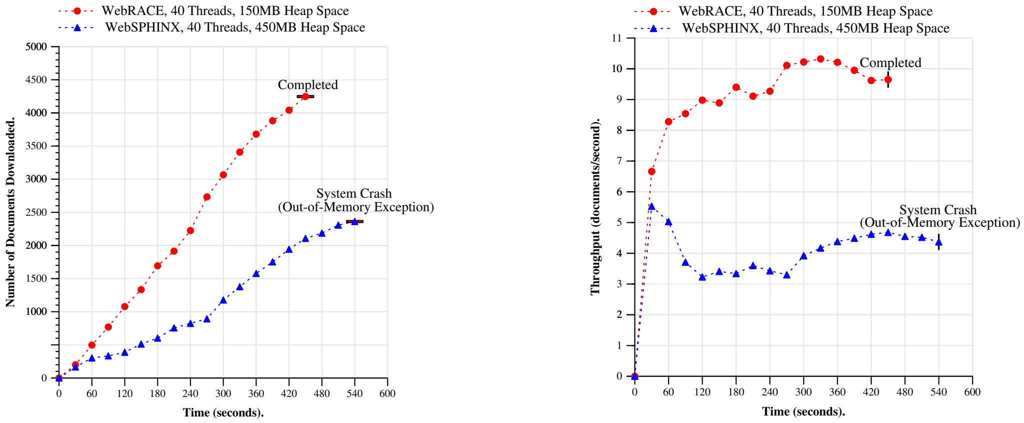
[1] This kind of crawler would be called, according to the terminology of [16], *a static-assignment firewall-mode crawler.*

to a larger set of valuable pages. Subsequent research in [24] suggests that by using an additional back-crawl phase (i.e. crawling towards the parents rather than children) before proceeding to the crawling phase can further improve the precision of focused crawlers by 50-60%. Although focused crawlers are able to build specialized Web indices with significant savings in hardware and network resources, they are not significantly useful as general purpose crawlers since the latter are interested in eventually exploring a large segment of the WWW graph.

iv) Peer-to-peer crawlers: Search engines usually rely on expensive centralized infrastructures comprised of clusters of workstations with terabytes of disk storage and many gigabytes of main memory. Although such infrastructures are feasible, they are not very scalable as many hundreds of gigabytes of new data are added to the WWW on a daily basis. A peer-to-peer approach for addressing the scalability problem is adopted by the Grub Crawler [31], which deploys a peer-to-peer infrastructure at which peers donate their computer's unused bandwidth to help probing the Web for pages that have changed. This reduces the burden of the actual crawler since it does not need to continuously examine which pages have changed and, therefore, can cope with information that changes frequently. A similar approach is followed by the YouSearch project [9] at IBM Almaden, which proposes the deployment of transient peer crawlers to build a search engine that provides "always-fresh" content. The main idea in YouSearch is that each user using the service contributes its host to become a transient crawler. In effect, this results to a network of transient crawlers in which each crawler maintains an "always-fresh" snapshot of a pre-specified list of Web resources. Each crawler also sends a compact index of its crawl (i.e. a bloom filter), to a centralized component at regular intervals. This helps the system redirect some user's query to the most relevant crawler, i.e. to the crawler that has content relevant to the query, rather than flooding the network with queries.

8.3 Performance Assessment

To assess the performance of WebRACE we compare it with the crawler of WebSPHINX, since both systems are written in JAVA and support multithreading. We could not compare WebRACE with many commercial (e.g. Overture, Google or MSN) or research crawlers [41, 16, 12], as none of them was publicly available. Furthermore, we did not compare WebRACE



DOWNLOADS of WebRACE and WebSPHINX
crawling the homepages of the 10 UC Campuses (3 levels).

THROUGHPUT of WebRACE and WebSPHINX
crawling the homepages of the 10 UC Campuses (3 levels) .

Figure 11: Crawling the 10 UC Campuses using WebRACE and WebSPHINX.

to existing single-threaded crawlers, such as the unix-based *wget crawler*, since such a comparison would not be fair.

To compare WebRACE and WebSPHINX quantitatively, we deployed both crawlers with a static seed list, that is without injecting new crawl requests at runtime. We made this choice because WebSPHINX does not provide any explicit mechanism to cope with runtime request scheduling. Our benchmark ran on a 2.0GHz Intel Pentium IV, using 512MB RAM, running Windows XP and Sun's Java 1.4.2 runtime. Our seed list included the homepages of the 10 University of California campuses, and our crawl spanned three levels for each entry in the seed list. We configured both crawlers to use breadth-first traversal using 40 threads and restricted them to download only content that did not exceed 1MB. We also configured WebRACE's maximum heap space (i.e. the maximum space allocated by the runtime) to 150MB. On the other hand we allowed WebSPHINX to use a heap of up to 450MB, since it consumes extraordinary amounts of memory.

As we can see in figure 11, the WebRACE crawler outperformed the WebSPHINX crawler as it accomplished the crawl of 4247 objects at an average download rate of 9.3 pages/second. The WebSPHINX crawler on the other hand did not manage to accomplish the crawling task as it crashed with an `Out-of-Memory` exception. We believe that this is caused by the fact the WebSPHINX crawler maintains several in-memory structures. Another observation is that the throughput of the WebSPHINX crawler is consistently low (≈3.9 pages/second). This happens even at the first 120 seconds, where the crawler does not utilize its whole available heap space, which implies that the crawler would not be able to perform any better even in the presence of more main memory.

8.4 Remarks

In our work, we address the challenge of designing and implementing modular, open, distributed, and scalable intermediary infrastructures, using Java. Our effort differs from other systems in a number of key issues: (i) Our design is established upon a set of XML grammars that define the information and meta-information collected, generated and exchanged between the various modules of our system. The employment of XML makes it easier to decouple information retrieval, storage and processing from content publishing and distribution. Furthermore, it enables us to describe easily new services, terminal devices, and information sources, without altering the basic design of our infrastructure. Finally, it exports the operations "executed" by the engine to the system-programming level, thus making possible the implementation of different scheduling and resource-management policies according to differentiated service-levels provided to individual users or services. (ii) eRACE consists of modules that communicate via messages and events; therefore, it is relatively easy to distribute these modules to different machines, achieve distributed operation and scalable performance. (iii) Instead of making eRACE a totally generic infrastructure, we developed and optimized modules which provide a functionality necessary nowadays in most intermediary systems: a "user-driven" high-performance crawler, an object cache, and a filtering processor. These, performance-critical modules are multithreaded and distributed thanks to the employment of Java and of distributed data structures in Java. (iv) The development of new services on top of eRACE is supported through the definition of new XML profiles inserted at the information architecture of eRACE. Additional programmability is provided by the employment of the mobile-agent programming paradigm, which supports mobility and disconnected operation of end-users as well. (v) Although a large number of papers have been published on Web crawlers [39, 33, 17, 13, 45, 41, 14], and Internet middleware, the issue of incorporating flexible, scalable and user-driven crawlers in middleware infrastructures remains open. This issue is addressed in the context of WebRACE.

9 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. WebRACE consists of a high-performance, distributed Web crawler, a filtering processor, and an object cache, written entirely in Java. In our work, we addressed the challenges arising from the design and development of WebRACE. We described our design and implementation decisions, and various optimizations. Furthermore, we discussed the advantages and disadvantages of using Java to implement the crawler, and presented 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. The combination of techniques such as, multithreading, caching the crawling state, and the employment of distributed data stuctures in Java, improved the scalability and performance of WebRACE.

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.

References

[1] Open Pluggable Edge Services. http://www.ietf-opes.org.

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

[3] Open Pluggable Services Working Group Charter. http://www.ietf.org/html.charters/opes-charter.html, October 2003.

[4] R. Aiken, M. Carey, B. Carpenter, I. Foster, C. Lynch, J. Mambreti, R. Moore, J. Strasnner, and B. Teitelbaum. Network Policy and Services: A Report of a Workshop on Middleware, 2000. RFC 2768, IETF. http://www.ietf.org/rfc/rfc2768.txt.

[5] 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 Conference on Advanced Multimedia Content Processing, AMCP '98, Lecture Notes in Computer Science*, pages 194–207. Springer Verlag, 1999.

[6] R. Barrett and P. Maglio. Intermediaries: New Places for Producing and Manipulating Web Content. *Computer Networks and ISDN Systems*, 30(1–7):509–518, April 1998.

[7] R. Barrett and P. Maglio. Intermediaries: New places for producing and manipulating Web content. In *Proceedings of the Seventh International World Wide Web Conference (WWW7)*, 1998.

[8] R. Barrett and P. Maglio. Intermediaries: An approach to manipulating information streams. *IBM Systems Journal*, 38(4):629–641, 1999.

[9] M. Bawa, R.J. Bayardo, S. Rajagopalan, E. Shekita. "Make it Fresh, Make it Quick – Searching a Network of Personal Webservers". In Proc. of the 12th Int. World Wide Web Conference, WWW-2003, May 2003, Budapest, Hungary

[10] E. Brewer, R. Katz, E. Amir, H. Balakrishnan, Y. Chawathe, A. Fox, S. Gribble, T. Hodes, G. Nguyen, V. Padmanabhan, M. Stemm, S. Seshan, and T. Henderson. A Network Architecture for Heterogeneous Mobile Computing. *IEEE Personal Communications Magazine*, 5(5):8–24, October 1998.

[11] E. A. Brewer. Lessons from Giant-Scale Services. *Internet Computing*, 5(4):46–55, July-August 2001.

[12] 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.

[13] 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.

[14] M. Chau and H. Chen. Personalized and Focused Web Spiders. In N. Zhong, J. Liu, and Y. Yao, editors, *Web Intelligence*, chapter 10, pages 197–217. Springer-Verlag, February 2003.

[15] Yih-Farn Chen, Huale Huang, Rittwik Jana, Trevor Jim, Matti Hiltunen, Sam John, Serban Jora, Radhakrishnan Muthumanickam, and Bin Wei. iMobile EE: an enterprise mobile service platform. *Wireless Networks*, 9(4):283–297, 2003.

[16] J. Cho and H. Garcia-Molina. Parallel Crawlers. In *Proceedings of the Eleventh International World-Wide Web Conference*, pages 124–135. ACM Press, May 2002.

[17] 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.

[18] I. Cooper, I. Melve, and G. Tomlinson. Internet Web Replication and Caching Taxonomy, January 2001. IETF, RFC 3040, http://www.ietf.org/rfc/rfc3040.txt.

[19] 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.

[20] M. Dikaiakos, A. Stassopoulou, L. Papageorgiou. Characterizing Crawler Behavior from Web Server Access Logs. In E-Commerce and Web Technologies. Proceedings of the 4th International Conference on Electronic Commerce and Web Technologies (EC-Web 2003), K. Bauknecht, A. Min Tjoa and G. Quirchmayr (Eds.), Lecture Notes in Computer Science series, vol. 2738, pages 369-378, Springer, September 2003

[21] M. Dikaiakos. Intermediary Infrastructures for the World-Wide Web. *Computer Networks*, 2004. To appear.

[22] M. Dikaiakos and D. Gunopulos. 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.

[23] M. Dikaiakos, M. Kyriakou, and G. Samaras. Performance Evaluation of Mobile-Agent Middleware: A Hierarchical Approach. In G. P. Picco, editor, *Proceedings of the 5th*

International Conference on Mobile Agents (MA 2001), volume 2240 of *Lecture Notes in Computer Science*, pages 244–259. Springer, 2002.

[24] M. Diligenti, F.M. Coetzee, C.L. Giles, and M. Gori. Focused Crawling Using Context Graphs. In *Proceedings of the 26th VLDB Conference*, pages 527–534, 2000.

[25] F. Douglis, T. Ball, Y.-F. Chen, and E. Koutsofios. The AT&T Internet Difference Engine: Tracking and Viewing Changes on the Web. *World Wide Web*, 1(1):27–44, January 1998.

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

[27] A. Fox, I. Goldberg, S. Gribble, D. Lee, A. Polito, and E. Brewer. Experience with Top Gun Wingman: A Proxy-based Graphical Web Browser for the 3Com PalmPilot. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, pages 407–426, 1998.

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

[29] 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)*, pages 319–332. The USENIX Association, 2000.

[30] 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, S. Ross, and B. Zhao. The Ninja Architecture for Robust Internet-scale Systems and Services. *Computer Networks*, 35:473–497, 2001.

[31] Grub - "help crawl it all", LookSmart. `http://www.grub.org/`

[32] J. Han, M. Kamber "Data Mining: Concepts and Techniques". Morgan Kaufmann, San Francisco, California, 2001.

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

[34] 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.

[35] A. Joshi. On proxy agents, mobility, and web access. *Mobile Networks and Applications*, 5:233–241, 2000.

[36] R. Koblick. Concordia. *Communications of the ACM*, 42(3):96–99, March 1999.

[37] Horizon Systems Laboratory. *Mobile Agent Computing. A White Paper*. Mitsubishi Electric ITA., January 1998.

[38] P. Maglio and R. Barrett. Intermediaries Personalize Information Streams. *Communications of the ACM*, 43(8):96–101, August 2000.

[39] 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.

[40] D. Milojicic. Internet Technology. *IEEE Concurrency*, pages 70–81, January-March 2000.

[41] M. Najork and A. Heydon. High-Performance Web Crawling. Technical Report 173, Compaq Systems Research Center, September 2001.

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

[43] M. Perkowitz and O. Etzioni. Towards adaptive Web sites: Conceptual framework and case study. *Artificial Intelligence*, 118:245–275, 2000.

[44] 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.

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

[46] H. Rao, Y. Chen, D. Chang, and M. Chen. iMobile: A Proxy-based Platform for Mobile Services. In *The First ACM Workshop on Wireless Mobile Internet (WMI 2001)*, pages 3–10. ACM, 2001.

[47] H. Rao, Y. Chen, and M. Chen. A Proxy-based Web Archiving Service. In *Middleware Symposium*, 2000.

[48] G. Samaras, M. Dikaiakos, C. Spyrou, and A. Liverdos. Mobile Agent Platforms for Web-Databases: A Qualitative and Quantitative Assessment. In *Proceedings of the Joint Symposium ASA/MA '99. First International Symposium on Agent Systems and Applications (ASA '99). Third International Symposium on Mobile Agents (MA '99)*, pages 50–64. IEEE-Computer Society, October 1999.

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

[50] 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.

[51] Franois Yergeau, Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible Markup Language (XML) 1.0 (Third Edition) W3C Recommendation. Technical Report REC-xml-20040204, World-Wide Web Consortium, http://www.w3.org/TR/2004/REC-xml-20040204/, February 2004.

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

[53] D. Zeinalipour-Yazti and M. Dikaiakos. High-Performance Crawling and Filtering in Java. Technical Report TR-01-3, Department of Computer Science, University of Cyprus, June 2001.

[54] D. Zeinalipour-Yazti and M. Dikaiakos. Design and Implementation of a Distributed Crawler and Filtering Processor. In A. Halevy and A. Gal, editors, *Proceedings of the Fifth International Workshop on Next Generation Information Technologies and Systems (NGITS 2002)*, volume 2382 of *Lecture Notes in Computer Science*, pages 58–74. Springer, June 2002.