

Crawllets: Agents for High Performance Web Search Engines

Prasanna Thati, Po-Hao Chang, and Gul Agha

University of Illinois at Urbana-Champaign
{thati,pchang2,agha}@uiuc.edu

Abstract. Some of the reasons for unsatisfactory performance of today's search engines are their centralized approach to web crawling and lack of explicit support from web servers. We propose a modification to conventional crawling in which a search engine uploads simple agents, called *crawllets*, to web sites. A crawllet crawls pages at a site locally and sends a compact summary back to the search engine. This not only reduces bandwidth requirements and network latencies, but also parallelizes crawling. Crawllets also provide an effective means for achieving the performance gains of personalized web servers, and can make up for the lack of cooperation from conventional web servers. The specialized nature of crawllets allows simple solutions to security and resource control problems, and reduces software requirements at participating web sites. In fact, we propose an implementation that requires no changes to web servers, but only the installation of a few (active) web pages at host sites.

1 Introduction

Web search engines are being widely used for searching information on the web. Although they provide a valuable service, experiments have shown that their performance is far from satisfactory [11]. Most search engines cover only a small fraction of the web, and the mean time between the modification (or creation) of a page and the time it is re-indexed by a search engine is several weeks long.

This under-performance is largely due to the centralized nature of search engine design, and the lack of cooperation between search engines and web servers. Today's search engines crawl hundreds of millions of web pages across the network, all from one place (or at most from a handful of sites), generating one network transaction per page. This approach does not scale well. Further, most of the current web servers do not distinguish crawler requests from regular requests, despite the fact that there is a variety of site-specific information that can be made available to significantly improve the performance of search engines [2]. Given the scale of the web, any solution to these problems that requires modifications to web servers or excessive support from web sites, such as significant software installations, would be impractical. As usual, by a web server we mean servers such as Apache and IIS (Internet Information Service) that serve HTTP

requests, and by a web site we mean the system including the machine and the software platform on which a web server is executed.

We propose a solution that uses mobile agent technology to modify the way conventional search engines crawl the web. Our approach is not only scalable but also achieves the performance gains of web servers that export site specific information. It requires minimal modification to the search engines, and just the installation of a few active web pages (e.g. CGI programs or ASPs) at participating web sites. The specialized nature of our agents offers simple solutions for securing both web sites and agents against malicious attacks. Our solution does not assume unanimous cooperation from all web sites, instead provides the flexibility of varying degrees of cooperation from web sites, and seamlessly integrates conventional crawling of non-cooperating sites. We will also argue that there are strong incentives for certain types of web sites to support our scheme.

The layout of the rest of this paper is as follows. In Sect.2, we take a closer look at the design of conventional search engines and discuss the problems in greater detail. In Sect.3, we discuss some of the solutions that have been proposed to alleviate these problems, and identify their deficiencies. In Sect.4, we present an overview of our solution, and discuss how it achieves our goal without introducing any new problems. In Sect.5, 6 and 7 we describe its details, and in Sect.8, we present experimental results that demonstrate the performance gains. We conclude the paper in Sect.9.

2 A Critical Analysis of Conventional Search Engine Design

The architecture of conventional search engines roughly consists of three components - a crawler, an indexer, and a searcher. The crawler starts with a set of seed URLs and repeatedly downloads pages, extracts hyperlinks from the pages, and crawls the extracted links. Since web pages may change, the crawler revisits previously crawled pages once in a while. The crawled pages are stored in a repository which is accessed by the other components. The indexer parses each downloaded page to generate compact summaries, and constructs indices that map individual words to the page summaries they occur in. The searcher uses these indices to respond to search queries from users.

We identify some problems with the crawler component that affect the performance of search engines.

Centralized Architecture: The crawling strategy is highly centralized and does not scale well. Usually, an HTTP request and reply are generated for every page downloaded by the crawler. Each of these requests and replies requires a separate TCP connection. Given that the web today has well over 1 billion publicly indexable pages [11], the latency involved in establishing these connections quickly adds up. Further, the estimated amount of data in all the web pages is of the order of tens of terabytes and continues to grow. Since a search engine has to frequently re-crawl web pages to account for any changes, the network

bandwidth required is tremendous. Moreover, all the downloaded pages are processed locally to generate page summaries, which requires a lot of storage and processing power.

It is possible to meet these formidable challenges by using extra hardware. In fact, Google uses multiple crawlers to open hundreds of concurrent connections with web servers and downloads hundreds of kilobytes of data per second [13]. It uses thousands of networked PCs to process the downloaded pages. The problem with this approach is the accompanying hardware and software maintenance cost, including failures, power consumption, and hardware and software upgrades.

Inaccurate scheduling policies: Different web pages have different rates of change, extent and types of change, and relative importance. Since a search engine has only a finite amount of computational resources, different strategies to revisit pages may result in different levels of freshness and importance of results it returns for search queries. The reader is referred to [4] for a good discussion on this problem.

Several scheduling strategies for crawling have been proposed in the recent past. In [5], the authors propose URL ordering schemes based on certain importance metrics for pages, that can be used to refetch important pages more frequently. The scheduling algorithm presented in [3] partitions pages according to their popularity and rate of change, and allocates resources to each partition using a custom strategy. The main drawback of these approaches is that they either do not account for or make unreasonable approximations of significant parameters such as frequency and type of page changes. For instance, the algorithm in [3] approximates page changes as memoryless Poisson processes, but a large class of pages such as those of newspapers and magazines change on a periodic schedule which is not Poisson. Bad estimation of these additional parameters increases the probability that a crawler fetches a page that has not changed since its last visit. Indeed, experimental data [2] show that almost 50-60% of revisits by a naive crawler may be unnecessary. In summary, the parameters required for efficient scheduling policies have a very broad range and are highly dependent on the pages being crawled and the web sites that host them. Given the variety of pages and sites, it is difficult to predict these quantities accurately.

Incomplete and Unnecessary Crawling: Due to resource constraints search engines crawl most web sites only to a certain depth. This contributes to incompleteness of crawling. Moreover, not all pages fetched by a search engine may be relevant. This is especially true for special purpose search engines, such as the media-specific engines which only look for specific types of documents (e.g. GIF or MP3 files). The only way a conventional crawler can discover these files is by finding links to them from other HTML documents it crawls. Obviously, fetching these additional pages is a costly overhead.

3 Related Work

The lack of customized interaction between web servers and search engines is addressed in [2]. The authors suggest modifications to the interaction protocol

so that web servers can export meta-data archives describing their content. This information can be used by search engines to improve their crawling efficiency. The main drawback of this approach is that it requires modifications to the web servers. The enormity of the number of web servers raises some serious concerns about the deployment of such changes. Further, all the search engines and web servers have to agree on the syntax and semantics of languages used to describe the meta-data.

In [9], as an alternative to the centralized architecture of search engines, a push model is proposed in which individual web sites monitor changes to their local pages and actively propagate them to search engines. A simple algorithm is used to batch multiple updates into cumulative pushes. This reduces the load on search engines by distributing it amongst several web sites. However, it introduces a new challenge, namely the task of coordinating the pushes from millions of web sites. Bad coordination could result in too many simultaneous pushes to the search engine. Further, mechanisms to control excessive pushes from overzealous and non-cooperating web sites have to be devised. This would require extensive infrastructure, and co-ordination that involves too many principals.

The Harvest project [1] is another promising proposal for a distributed infrastructure for indexing and searching information on the web. The Harvest system provides an integrated set of tools to gather, index, search, cache and replicate information across the Internet. The distributed nature of information gathering, indexing and retrieval makes Harvest very efficient and scalable. However, as in the case of push model, the main drawback of this system is that it requires extensive software installations and agreed standards across the Internet. For instance, multiple sites have to agree to host Harvest system components that cooperate with each other. In our opinion, such schemes are up against an enormous inertia.

4 Our Approach: Dispatching Crawlets to Web Sites

The basic idea behind our approach is quite simple (see Fig.1). Instead of crawling pages at a web site across the network the search engine uploads an agent, called the crawlet, to the site. The crawlet crawls pages at the site locally and sends back the results in a custom format.

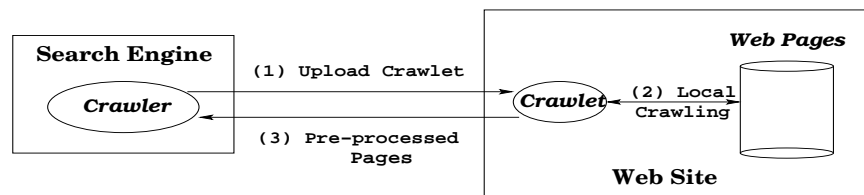


Fig. 1. A distributed approach to web crawling.

Using crawlets has a number of advantages. Typically, only one network transaction is required for crawling an entire web site, thus avoiding the latency in establishing a separate connection per page. Moreover, downloading large collections of compressed pages per transaction would result in significant bandwidth savings. Further, the crawlet may send back pre-processed page summaries instead of raw pages, further reducing bandwidth requirements. This also helps shift computational load from search engines to web sites with sufficient resources. Given the fact that there are about 3 million publicly indexable web servers [11], shifting small loads to many servers would cumulatively reduce significant computational load at the search engine. Also, since the search engine can upload crawlets parallelly to several web sites and receive the results asynchronously the crawling time is reduced by several orders of magnitude. All these savings would translate to reduced hardware requirements, reduced hardware and software maintenance cost, increased web coverage, and increased freshness of the search results.

Crawlets can also achieve the performance gains of customized web servers such as those that export meta-data with site specific information. For instance, a crawlet can discover pages that have changed significantly and ship only them back to the search engine. Further, for media specific crawlers the crawlet can discover and ship only the relevant media files. In both these cases, although the total time spent on processing pages is the same, the savings in network transactions are significant. Thus, there is no need for web servers to export meta-data, and for search engines to use complex scheduling heuristics (which often rely on unknown parameters) for revisiting pages. Modifying web servers to export meta-data as in [2], is only a temporary solution and does not work well in practice. Any such change will have to wait for server updates before it can be used. In contrast, once there is an agreement on the execution environment for crawlets search engines can upload any clever or personalized foraging crawlets.

Crawlets are not general information retrieval agents such as those described in [14], which navigate through the network interacting and coordinating with other agents and services to find relevant information. Instead, they are very specialized agents which do not communicate with other agents and once uploaded to a site do not migrate any further. Moreover, due to the non-critical nature of crawling, crawlets typically do not require guarantees such as persistence or fault-tolerance. As a result they do not need a general purpose agent platform. We propose an implementation that requires no modifications to web servers, but only the installation of a few active web pages at participating sites.

There is a good incentive for certain types of web sites to support our scheme, especially the ones that are not regularly crawled and indexed by search engines, either because their contents change too frequently or they are considered unimportant. For instance, the search results returned by Google rarely contain pointers to (even the major) news sites. Nevertheless, security is major issue for both hosting web sites and crawlets. Hosts are to be protected from threats such as unauthorized disclosure or modification of information and denial of service attack. Dually, the crawlet has to be protected from unauthorized tampering of

its code and data. For instance a host can modify the crawler output in order to improve the number of hits to its pages in response to search queries.

Ideally, the security measures required should be simple and computationally inexpensive, lest they will neutralize the benefits of our scheme. As we will see in Sect.7, since crawlers require very limited and predictable access to system resources, security can be enforced by simple sandboxing mechanisms. Similarly there are computationally inexpensive techniques to protect a crawler from malicious hosts.

We recently discovered an independent work that proposes the idea of using mobile agents for efficiently crawling the web [10]. A standalone application is presented that allows one to launch crawler agents to remote web sites. The application also provides powerful querying and archiving facilities for examining and storing the gathered information. In our opinion, using this application to implement web search engines has several drawbacks. First, this approach does not intergrated well with conventional search engine design. Search engines would typically employ highly customized information storage and retrieval techniques specifically suited for their purposes, instead of those imposed by the application. Thus, the application is best viewed as a platform that supports small scale web search applications. In contrast, we present a tightly integrated implementation (see Sect.5) that involves a few simple modifications to the crawler component, and leaves ample room for search engine specific customizations. Second, security and fine-grained resource control issues that arise due to code migration, have not been satisfactorily addressed in [10]. The severity of these concerns is heightened by the fact that their agents may have multi-hop itineraries. In contrast, the specialized nature of our crawlers, such as their single hop itineraries, provides simple security and resource control solutions (see Sect.7). Third, the proposal in [10] imposes excessive software requirements on participating web sites, such as rule based inference engines and sophisticated communication mechanisms. These software installations are also heavy on the CPU, and may thus discourage web sites from participating. In contrast, a guiding principle of our design has been to require a minimal and light weight software infrastructure.

5 Modifications to the Search Engine

Our scheme requires modifications to only the crawler component of a search engine. To simplify the discussion, we only show modification to a naive crawler which does not employ sophisticated scheduling policies such as those mentioned in Sect.2. However, our discussions can easily be adopted to include them.

<code>extract(Q,h)</code>	: Returns URLs in queue <code>Q</code> that are at <code>h</code>
<code>dispatchCrawler(h,F,U)</code>	: Sends a crawler to <code>h</code> along with URL lists <code>U</code> and <code>F</code> . Returns the URLs crawled, and the outgoing links found by the crawler
<code>crawl(u)</code>	: Downloads the page pointed to by <code>u</code> and returns all the links in the page

```

FQ = TQ = emptyQueue
enqueue(FQ,seedURLs)
while (true)
    TQ = FQ; FQ = emptyQueue
    while (TQ not empty)
        u = dequeue(TQ)
        if (site(u) supports crawlets)
            U, F = extract(TQ,site(u)), extract(FQ,site(u))
            (s,o) = dispatchCrawlet(site(u),F,U)
            enqueue(FQ,s); enqueue(TQ,o-FQ)
        else
            l = crawl(u)
            enqueue(FQ,u); enqueue(TQ,l-FQ)
        end if
    end while
end while

```

The modified crawler shown above maintains two URL queues - FQ which contains the URLs that have already been crawled, and TQ which contains URLs that are yet to be crawled. Before downloading the page at a URL u , the crawler checks if the corresponding web site ($\text{site}(u)$) supports crawlets. Web sites that do, export an active page at a fixed relative URL, say `srch-eng/crwlt-ldr.cgi`. Thus, before downloading `http://osl.cs.uiuc.edu/foundry/index.html` the crawler contacts `http://osl.cs.uiuc.edu/srch-eng/crwlt-ldr.cgi`. If this fails it resorts to conventional crawling. Else, it uploads a crawlet to the web site. Details of interactions with the active pages are described in Sect.6.

The crawlet carries two lists of URLs that belong to the remote site: U which contains seed URLs, and F which contains URLs that have already been crawled. These lists are extracted from TQ and FQ the obvious way. The crawlet crawls pages at its host in the conventional style, except that it does not crawl a newly discovered link if it points to a remote site. The crawlet ships back a list of URL/page-summary pairs that correspond to crawled links and a list of outgoing links it did not crawl. Further details about crawlet implementation and related issues such as security are discussed in Sect.7.

The crawlet may not be able to discover all the pages at the web site in its first visit. For example, consider the graph of pages shown in Fig.2. An arrow between two pages means that the source page contains one or more links to the destination page. Suppose the crawler only knows the URL of w_{11} . The crawler dispatches a crawlet with w_{11} as the seed URL. The crawlet replies back with w_{11} and w_{12} as the URLs crawled, and w_{21} and w_{32} as the outgoing URLs found. Note that the crawlet cannot reach pages w_{13} and w_{14} in this visit. But later when the crawler gets to crawl w_{22} , it discovers the URL of w_{13} . In the second visit the crawlet can crawl the remaining pages. In general, to avoid multiple visits to a site, the crawler may dispatch a crawlet with a larger set of seed URLs that were discovered in its previous iterations.

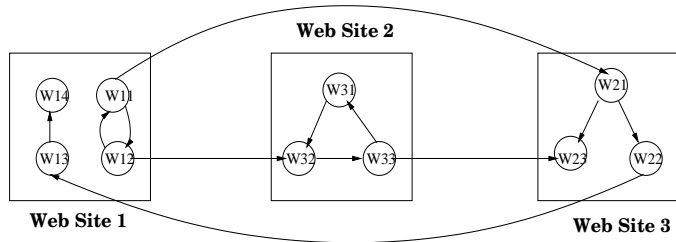


Fig. 2. A graph of web pages residing at three web sites

In the crawler algorithm shown above, once the crawlet is uploaded to a web site the crawler does not crawl any further until it hears back from the crawlet. Experimental results (see Sect.8) show that even in this simple case the crawler is significantly faster than the conventional one. The crawling can be readily parallelized to obtain further speed ups by dispatching multiple crawlets to different sites and asynchronously waiting for their replies.

6 Uploading Crawlets

We now discuss the protocol between search engines and the (active pages at) web sites, that is used to upload and execute crawlets. The security concerns that arise include mutual authentication between the search engine and the web site to prevent malicious masquerading, and the integrity and confidentiality of the transaction data. It is reasonable to expect the web site to execute the crawlet under certain resource constraints. The web site may adopt different resource allocation policies for different search engines, which may also vary with time depending on local system state. It is useful to convey these resource constraints to the search engine so that it may upload a customized crawlet.

Following is a simple protocol built on top of HTTP (see Fig.3) that implements these requirements. It uses public key cryptography for authentication, and shared key encryption for secure transmission of data.

Request 1 : An HTTP request from the search engine to the web site. The request payload contains a randomly generated session key K and is encrypted with the secret-key S_s of the search-engine.

Reply 1 : The crawlet receives a reply containing a preamble confirming that the receiver understands the protocol, and an authentication key and a time nonce that should be used as a ticket for subsequent transactions. The reply also specifies the resource limits (such as CPU time, memory and disk space) that would apply to the crawlet. The payload is encrypted twice, first with the secret key S_w of the web site, and next by the session key K .

Request 2 : An HTTP request from the search engine. The payload consists of the authentication key, nonce, and the crawlet along with its data. It is encrypted with the session key K .

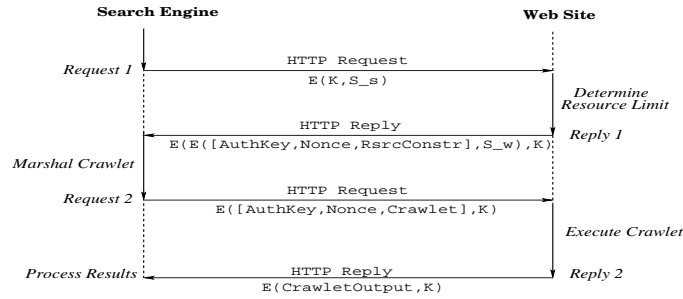


Fig. 3. The protocol used to upload and execute crawllets.

Reply 2 : An HTTP reply that contains the output that the crawllet writes to its standard output. The output stream is not interpreted by the web site. The stream can also be interrupted by an error message, which could be because of crawllet errors, or host-initiated crawllet abortion due to violation of security or resource constraints. The stream is encrypted with the session key K .

Due to the non-critical nature of the crawllets and the information they gather, typically there are no strict fault-tolerance or consistency requirements on the uploading and execution of crawllets. The protocol described above has been kept simple to minimize software requirements at web sites. However, it can be extended in several useful ways. The web site can furnish details about its local platform such as the type of its HTTP server, operating system, and the organization of web pages in the local file system. Such information can be used by the crawler to dispatch customized crawllets that crawl pages much more efficiently. There could also be a more elaborate negotiation between the crawler and the web site for resources. One possibility is to have the web sites ‘sell’ their computational resources in exchange for increased importance that the search engine associates with its pages while processing search queries.

7 Executing Crawllets

Once the crawllet is uploaded it is executed by its host (active pages) in a controlled environment. Following is a simple algorithm for crawllets.

- `crawl(u)` : Crawls the page pointed to by URL u and returns the list of URLs it points to and its pre-processed contents.
- `classify(l)` : Classifies the URL list l into two lists: one containing URLs at the local site, and the other the remaining.

```

FQ, TQ, OQ = previouslyCrawledURLs, seedURLs, emptyQueue
while (TQ not empty)
    u = dequeue(TQ); (c,l)= crawl(u)
    enqueue(FQ,u)
    (o, i) = classify(l)
    o = o - OQ
    output((u,c),o)           // write to stdout
    enqueue(OQ, o), enqueue(TQ,i-FQ)
end while

```

The crawlet crawls local pages as usual by making HTTP requests. If the web server is not already optimized for such local requests the host can intercept these requests and directly access the file system. Of course, this only works for static HTML pages and not for dynamically generated ones, and it also requires the host to know the mapping between URLs and path names which may vary from web server to web server. Also, note that the crawlet periodically writes partial results to its standard output, which is continuously redirected back to the search engine by the host. This not only saves storage space for the crawlet, but also is handy if the crawlet does not get to complete its execution.

We now discuss security measures for protecting both hosts and crawlets from malicious attacks. The security problem is considerably simplified since the crawlet visits only one host. Recall that, conventional security mechanisms break down for general mobile agent applications primarily because of their unrestricted mobility. The problem with unrestricted mobility is that hosts in the agent's itinerary need not trust each other. Even if an agent is initially signed by its source, an intermediate host can alter its code and state to make it malicious. It is generally difficult for a receiving host to determine if the agent has been tampered with. Similarly it is difficult for an agent to determine if its execution environment at a host is untampered. The current approaches for securing agents that travel more than one hop include sophisticated approaches such as carrying proofs of code safety [12], maintaining agent state appraisal [6], and maintaining path (itinerary) histories [16].

A single hop itinerary and limited computational facilities required by the crawlets greatly simplify the problem of securing the hosts. Well known techniques used in conventional settings without migrating code are sufficient. The authentication and secure loading of crawlets was described in Sect.6. During its execution the crawlet only needs permission to make HTTP requests to the local web server, and use specific folders in the file system as scratch space. It neither needs to communicate with other agents nor access other system resources. The host is secure if it grants only these permissions to the crawlet and enforces the resource limits negotiated with the search engine. These security measures are easily realized using the sandboxing technique [7].

It is also important to protect crawlets from malicious hosts which may tamper its output. Although there isn't much incentive for a host to not forward crawlet outputs to the search engine, it may want to modify output such as the pre-processed page contents to improve its popularity. However, this problem is

not unique to our scheme. Since web sites can distinguish search engine requests from others they can forge replies with as much ease in the conventional setting. In any case, the tampered data is not critical enough to cause irreversible damage.

The possibility of hosts tampering with crawllet output is inconvenient enough to look for prevention mechanisms. A simple idea is to secure the crawllet's output stream by encrypting data along with embedded keys. Shared key encryptions are computationally very cheap and a fresh key is generated every time a crawllet is uploaded. But this solution does not prevent the host from interfering with the execution of the crawllet, in particular with the encryption process itself. A well known solution to this problem is to encipher the encryption function itself [15]. Suppose the crawllet is to use the function f to encrypt its output, but f is to be kept a secret. The search engine transforms f to some other encryption function $E(f)$ that hides f . The program $P(E(f))$ that implements $E(f)$ is embedded in the crawllet. The host can therefore only learn about $P(E(f))$ which is applied to produce the encrypted output. The search engine decrypts this output to obtain $f(x)$. Thus, once a suitable candidate for $E(f)$ is known this strategy is straightforward and is computationally inexpensive. In summary, the simplicity of crawllets enables us to enforce security measures that are computationally inexpensive and do not neutralize the benefits.

We end this section with a brief note on strategies the search engine may adopt if the resources allocated by a web site are insufficient. The pages at a site can be logically organized as a tree or a forest based on the path components of their URLs. In the presence of insufficient resources the search engine can have its crawllet crawl only a few subtrees in the forest. The crawllet would simply treat any link not pointing to a page in the subtrees as an outgoing link. The number of pages and their total size in a large subtree typically vary very little with time and hence, based on previous experience, it is feasible to estimate the resources required to crawl them. If the estimates are inaccurate, and the crawllet is unexpectedly short of resources it can ship its state back to the search engine so that crawling can continue either in the conventional style or through another crawllet.

8 Performance Measurements

We now present experimental results that demonstrate the effectiveness of our approach. The primary focus of our experiments is to measure the performance gains of crawling a single site with different types of crawllets in comparison with conventional crawling. An important factor in the experiments is the choice of web sites. According to web statistics, the average number of pages per web site is about 500 [11]. Further, most of the sites host very few pages and only a small portion host a very large number of pages. So a web site with about a few thousand pages is a good choice. We conducted experiments on three such sites (see Table 1). However, the sites have considerably different properties such as the number and size of pages and their linkage structure. To control the

experimental environment we mirrored the sites onto a web server which does not host any other pages. This is essential to isolate our measurements from variations due to factors such as differences in hardware and unpredictable load at web servers due to the regular request traffic (that is not generated by the experiments).

Table 1. Properties of sites used in the experiment. **Engineering:** <http://www.engr.uiuc.edu>, **ACM:** <http://www.acm.uiuc.edu>, and **Computer Science:** <http://www.cs.uiuc.edu>.

	Engineering	ACM	Computer Science
<i>Pages Crawled</i>	1739	2070	536
<i>Outgoing Links</i>	570	2446	325
<i>Total Size (MBytes)</i>	15.38	3.14	2.37

The active pages that receive and execute crawlets were implemented using Microsoft ASP. These pages implement the protocol described in Sect.6. They assume that crawlets are implemented in Java. In general, such assumptions and details of the execution environment can be conveyed while uploading the crawlet. The pages use the built in sandboxing facility of Java runtime environment for security control.

As far as the search engine is concerned, only the crawler component is relevant for our purposes. We implemented two versions of the crawler: one that crawls in the traditional style, and the other that uses crawlets. Both the crawlers were implemented in Java. The conventional one runs a few crawling threads (which download pages) and parsing threads (which process the pages to extract links) in parallel. The other crawler dispatches crawlets to web sites using the protocol described in Sect.6. Java's object serialization and security features were used for this. To get a fair comparison, crawlets were implemented using the same crawling and parsing mechanisms as the conventional crawler. We experimented with 4 different types of crawlets which vary on how they crawl local pages - using HTTP requests or through the file system¹, and how they transmit the results back - compressed or uncompressed. The compression is implemented using `java.util.zip` package. The crawlets not only ship back raw unprocessed pages but also the URLs that these pages point to. This means that the crawlets which do not use compression will be shipping more data than in conventional crawling. We did this to get a fair comparison because after crawling a site the conventional crawler will have extracted these URLs which may then be used by other components of the search engine.

The web server we used is Microsoft IIS version 5.0. It was executed on a Pentium III 450 with 192 MB RAM and the server version of Windows 2000. To

¹ As mentioned in Sect.7, this is actually an optimization in the host execution environment. There is no change to the crawlet code.

get a fair comparison, the crawler was executed on a machine with exactly the same configuration. The experiments were conducted in two different settings: one with both the crawler and server executing on the same LAN, and the other with the two in different domains with an effective connection bandwidth of more than 1 Mbps. The available network bandwidth in both configurations was more than what was required by the crawler and crawlets. Thus, the difference is almost only in the network latencies. Further, while using crawlets there wasn't much difference in the measurements for the two configurations. This confirms the fact that network latencies are almost completely masked. So we present their results only for the WAN configuration.

We measured four parameters: total time taken to crawl a web site, number of bytes transferred between the crawler and the server, load on the crawler machine, and load on the web server machine. The total time taken for different configurations is shown in Fig.4. The data confirms a number of our speculations in the previous sections, which were primarily about the WAN configuration.

- All the four types of crawlets outperform the conventional crawler.
- The crawlets which compress their results further reduce the crawling time.
- The optimization of redirecting HTTP requests of a crawlet directly to the file system reduces the crawling time.

We observed an interesting phenomenon in the LAN configuration. The conventional crawler performs almost as good as crawlets (outperforming in some cases). This is because in a LAN network latencies are negligible and the extra time taken by crawlets for compression or shipping additional data is amplified. But this phenomenon is not very significant in real world because all the interactions while crawling the web are over the WAN. In summary, using crawlets reduces crawling time. This translates to reduced mean-time of revisiting pages, which in turn implies improved freshness of results for search queries.

Figure 5 shows the total number of bytes transferred between the crawler and the web site. As expected, for the crawlers that do not use compression the amount of data to be transferred is more than in the conventional case. This is especially amplified for sites with several small pages, each pointing to a number of other pages, as for the ACM site (Table 1). But this is easily remedied using the compressing crawlet. Since the pages are mostly plain text, compression has a huge impact. For example, for the ACM site the data transferred is reduced by more than 70%. These numbers can be improved further if the crawlet ships back only page summaries, although this means more computational load on its host.

Figure 6 shows the CPU load on the crawler and web server machines in different settings. In the interest of space we have shown the numbers for only the engineering site which hosts the largest amount of data amongst the three sites (Table 1). In all cases, most of the user time is spent on processing pages, while most of the kernel time is spent on I/O operations. The graph is composed of three parts one each for the crawler and web server machines, and another for the total load which is the sum of the first two parts.

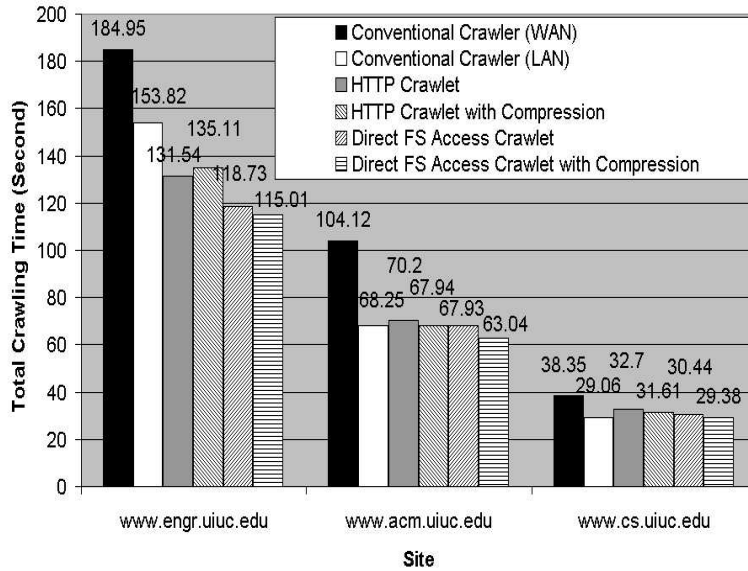


Fig. 4. Time taken to crawl a site.

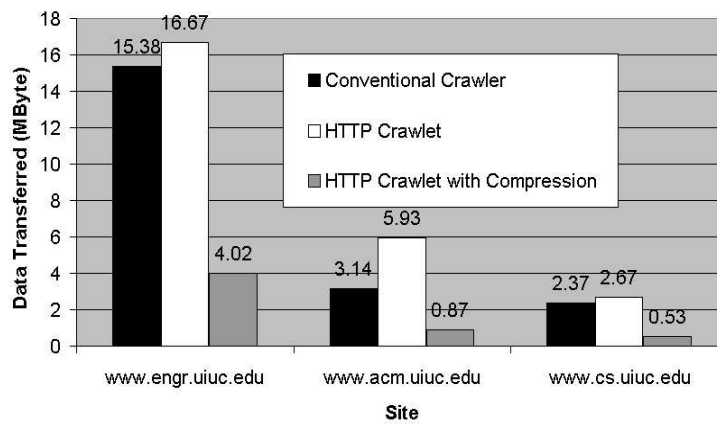


Fig. 5. Total number of bytes transferred between the crawler and a web site.

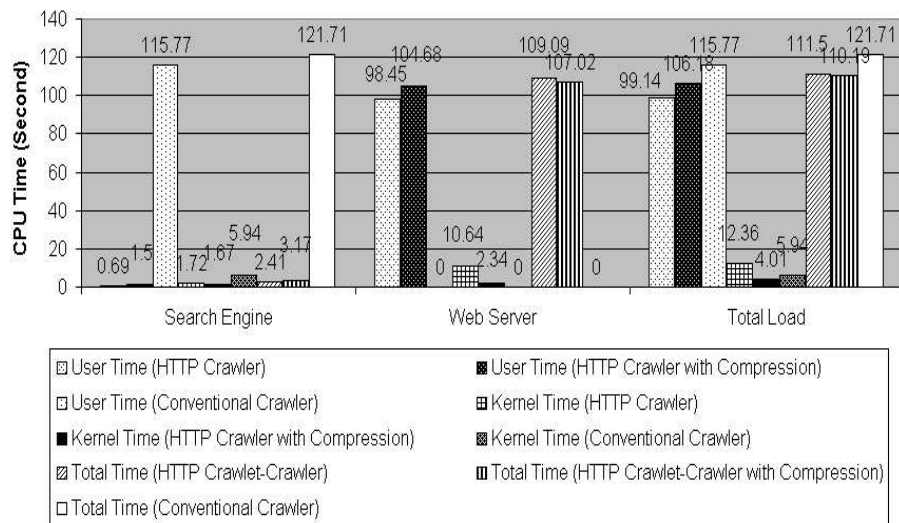


Fig. 6. CPU load on the crawler and web server machines.

It is obvious that there is significant shift in computational load from the crawler to the web server while using crawlets. But still, the combined load at the crawler and the web server is less than in the conventional case. This saving is primarily because of the reduction in number of network transactions required. Another observation is the effect of compression of the crawllet output. The increase in user time because of compression is less than the decrease in the kernel time because of reduced I/O operations, thus reducing the total CPU load. This is because the compression rate is very good for plain text.

Thus, we have demonstrated substantial reduction in crawling time, computational load at the search engine, and network transactions on using crawlets.

9 Conclusion

In this paper, we have proposed the use of mobile agents to improve the performance of web search engines. The performance gains translate to improved web coverage and freshness of search results. Implementations show that our scheme has minimal software requirements. In fact it only requires the installation of a few web pages at participating sites. This also greatly simplifies its deployment. The experimental results clearly demonstrate the performance gains. Due to its simplicity our proposal does not introduce new security concerns. Security can be enforced by simple conventional techniques which are computationally inexpensive. We believe that there is a strong incentive for several web sites to support our scheme, especially the ones that are rarely indexed by search en-

gines. Given that today's search engines cover only 12% of the web [11], there could be a large number of such sites.

An interesting research in the context of our work is on integrating agent platforms into web servers, such as in the WASP project [8]. Although promising, this idea has not yet gained a wide acceptance due to several reasons, including elaborate software installations required, security concerns, and incentive barriers. However, if accepted, our scheme can be readily integrated into such platforms in the form of services.

References

- [1] 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: Mosaic and the Web*, 1994.
- [2] Onn Brandman, Junghoo Cho, Hector Garcia-Molina, and Narayanan Shivakumar. Crawler-friendly web servers. In *Proceedings of the Workshop on Performance and Architecture of Web Servers*, 2000.
- [3] Brian Brewington. *Observation of changing information sources*. PhD thesis, Thayer School of Engineering, Dartmouth College, June 2000.
- [4] Brian Brewington and George Cybenko. Keeping up with the changing web. *IEEE Computer*, 33(5), May 2000.
- [5] Junghoo Cho, Hector Garcia-Molina, and Lawrence Page. Efficient crawling through url ordering. In *Proceedings of the 7th World Wide Web conference*, 1998.
- [6] William M. Farmer, Joshua D. Guttman, and Vipin Swarup. Security for mobile agents: authentication and state appraisal. In *Proceedings of the European Symposium on Research in Computer Security*, 1996.
- [7] J.S. Fritzinger and M. Mueller. *Java security*. Sun Microsystems, Inc., 1996.
- [8] Stefan Funfrocken. How to integrate mobile agents into web servers. In *Proceedings of the Workshop on Collaborative Agents in Distributed Web Applications*, 1997.
- [9] Vijay Gupta and Roy Campbell. Internet search engine freshness by web server help. In *Proceedings of Symposium on Applications and the Internet*, 2001.
- [10] Joachim Hammer and Jan Fiedler. Using mobile crawlers to search the web efficiently. *International Journal of Computer and Information Science*, 1(1), 2000.
- [11] Steve Laurence and Lee C. Giles. Accessibility of information on the web. In *Nature*, volume 400, July 1999.
- [12] G. Necula and P. Lee. Proof carrying code. In *ACM Symposium on Principles of Programming Languages*, 1997.
- [13] Lawrence Page and Sergey Brin. The anatomy of a search engine. In *Seventh International WWW Conference*, 1998.
- [14] Daniela Rus, Robert Gray, and David Kotz. Autonomous and adaptive agents that gather information. In *Proceedings of AAAI Workshop on Intelligent Agents*, 1996.
- [15] T. Sander and C.F. Tschudin. Protecting mobile agents against malicious hosts. In *Mobile Agents and Security*, 1998. LNCS 1419.
- [16] Giovanni Vigna. Protecting mobile agents through tracing. In *Proceedings of the 3rd ECOOP Workshop on Mobile Object Systems*, 1997.