

Distributed High-performance Web Crawlers: A Survey of the State of the Art

Dustin Boswell
dboswell [at] cs.ucsd.edu

December 10, 2003

1 Introduction

Web Crawlers (also called *Web Spiders* or *Robots*), are programs used to download documents from the internet. Simple crawlers can be used by individuals to copy an entire web site to their hard drive for local viewing. For such small-scale tasks, numerous utilities like `wget` exist. In fact, an entire web crawler can be written in 20 lines of Python code. Indeed, the task is inherently simple: the general algorithm is shown in figure 1. However, if one needs a large portion of the web (eg. Google currently indexes over 3 billion web pages), the task becomes astoundingly difficult.

2 Scaling Issues for Web Crawlers

There are inherent difficulties with crawling that could plague small-scale crawls as well, but are magnified when attempting large crawls. One problem is the unreliability of remote servers, which may accept incoming connections, but fail to reply, or reply at a snail rate of 10 bytes/second. Naturally, dns lookup, connecting to the site, and downloading must all be done asynchronously, or with adequate timeouts. Unfortunately, most software packages are not this flexible, and custom code is often needed.

Another trap to look out for is gigabyte-sized or even infinite web documents. Similarly, a web-site may have an infinite series of links (eg. “domain.com/home?time=101” could have a self-link

Figure 1: **General Web Crawling Algorithm**

```
Initialize:
  UrlsDone = {}
  UrlsTodo = {'yahoo.com/index.htm', ...}

Repeat:
  url = UrlsTodo.getNext()

  ip = DNSlookup( url.getHostname() )
  html = DownloadPage( ip , url.getPath() )

  UrlsDone.insert( url )

  newUrls = parseForLinks( html )
  For each newUrl
    If not UrlsDone.contains( newUrl )
      then UrlsTodo.insert( newUrl )
```

to “domain.com/home?time=102”, which contains a link to “...103”, etc...)¹

Not all domains wish to be crawled. The socially-accepted method by which web masters inform web crawlers which pages should not be crawled is by placing a “robots.txt” file in their root directory. Every new url must be checked against this list before being downloaded. When crawling a single domain, this is simple enough, but for a large crawl over millions of

¹And deciding to ignore dynamic pages results in a lot of skipped pages, while not completely fixing the problem.

domains, this is yet another data structure that must be built into the system. (It would be a filter inside the `UrlsTodo`.)

A related issue is that crawlers are expected to be *polite* and download pages from a domain at a reasonable rate. While human users click through a site at about a page every few seconds, a fast crawler trying to get thousands of pages a minute could bog down or crash small sites - similar to how a denial of service attack looks. In fact, web crawlers often set off intrusion alarms with their sudden surge of traffic. More generally, Google said it best that “running a web crawler generates a fair amount of phone calls.”

In addition to these logistical and software issues, there are hardware constraints as well. An obvious question is where to store all the web pages that have been downloaded. At an average page size of 20KB, a billion documents would require 20,000 Gigabytes. While pages can be compressed and/or stripped of markup, this only buys a factor of 10 or so. Clearly, multiple computers (perhaps hundreds or even thousands) are needed to distribute such a storage load. However, for this paper, we ignore the storage aspect, and focus on the rest of the system.

Another bottleneck is memory. Crawling the entire internet requires managing billions of urls, which won't all fit in memory, and so must be partially stored on disk. As a consequence, great care must be taken to avoid random disk accesses, or else that can become a bottleneck.

To maximize performance and avoid hardware bottlenecks, web crawlers are distributed across multiple machines (practically less than 10) in a fast local area network.

To download a billion pages in one year, a crawler must sustain a rate of 32 pages/second. However, search engines must also *recrawl* pages to obtain the most recent version, which amplifies the need for speed.

3 Previous Work

As large-scale crawler writing is difficult, and requires a decent amount of hardware and network speed, good papers are not abundant in the literature.

The largest, most successful systems are probably by search engines and other corporations, where details are not often given.

An exception is the well-documented *Mercator* project ([NH]), done at Compaq (and used by the AltaVista search engine). The initial crawler used by *Google* in 1998 is described in [BP98]. Various portions of the *Internet Archive* crawler are given in [Bur97].

The *UbiCrawler* project ([BCSV02]) is a high-performance crawler whose primary focus is distribution and fault tolerance. Another crawler with good implementation details is a project done at *Polytechnic University* ([SS02]). A crawler done at *Kasetsart University* in Thailand ([KaS]) achieved a very fast download rate for local `.th` domains. A paper done by *Cho and Garcia-Molina* ([CGM02]) thoroughly describes the design decisions of distributing and coordinating workload across different machines.

WebFountain [EMT01] is a large project at IBM currently being developed. The *WebRace* crawler ([ZYD]) is described as part of a larger “Retrieval, Annotation, and Caching Engine”.

In the following sections we will describe each of the components of a distributed web crawler in more detail, first setting up the problem, and then describing how previous systems attacked it.

4 Web Crawler Design

4.1 Distribution

The four critical components to a web crawler are:

- `UrlsTodo` data structure
- DNS lookup
- HTML download
- `UrlsDone` data structure

Given that we wish to distribute the workload across multiple machines, the first decision is how to divide and/or duplicate these pieces in the cluster. The natural unit of work is the url. However, since pages have links to many other pages, there is

still the need for inter-machine coordination so that pages are not downloaded multiple times.

Google's approach was to have a centralized machine acting as the url-server, with 3 other dedicated crawling machines, which only communicated directly to the url-server. As different machines have different functions, we call this a heterogeneous structure. Since they did not distribute the url data structures, this design would probably not scale well past the 24 million urls in their crawl.

Instead, Mercator uses a homogenous fully-distributed approach, where each machine contains a copy of each functional component. The url-space is divided into n pieces (one for each machine) by hashing the url's domain, and assigning that url to machine $(\text{hash}(\text{domain}) \% n)$. Thus, each machine is completely in charge of a subset of the internet. When links are found to a url outside that subset, they are forwarded to the appropriate machine. They report that since 80% of links are relative, a large majority of urls don't need to be forwarded.

The crawler built at Polytechnic has a system similar to Google's, where a centralized manager distributes url's to one of eight machines dedicated to crawling. They acknowledge, however, that to distribute the manager they could use a domain-based hashing system similar to Mercator, where managers would have to fully communicate. The result would be a heterogeneous tiered architecture.

Distribution based on mod'ing the domain hash-value by the number of machines is a *static* assignment function. In the event a machine goes down, or one wants to be added there is not much that can be done. The machines could possibly re-synchronize and notice the new number of machines, but this would require all url's in the system to be re-hashed and probably moved. Instead, UbiCrawler uses *consistent hashing* on the domain hash-value, over a homogenous cluster. To summarize, each domain implicitly has an ordered list of machines to try sending that url to. Each domain has a fixed random ordering, in such a way that domains are evenly distributed, even as the number of machines change. Since all machines have agreed on this scheme ahead of time, no re-synchronization is ever needed.

The system designed in [KaS] is interesting in that it distributes urls based simply on that url's hash value, spread over a homogenous cluster. Thus, urls from the same domain can be all over the system. This introduces a politeness coordination problem: how do you prevent different nodes from 'attacking' the same domain at the same time? To solve this, they introduce the concept of 'phase swapping', where at any instant, a machine is only crawling domains belonging to that phase (which is computed by hashing the domain). Machines are always in different phases, so no two machines ever download from the same domain at the same time. One wonders why they went to this effort when they could have just done domain hashing to begin with!

Since all other systems use a domain-based hashing scheme, they can address the issue of politeness on a per-machine basis independently. As there are other benefits to grouping urls by domain, like dns caching and link-locality, this appears to be most effective distribution function.

We now address the design of the other modules, which are assumed to be on a single machine.

4.2 Download module (per machine)

To be polite, a crawler should download from a given domain at a slow enough rate (say a page every 4 seconds). Urls from different domains should be downloaded in parallel to maximize network usage. Thus, attaining a rate of 25 pages/sec requires 100 concurrent connections. There are two general approaches: multiple threads, each taking 1 url at a time from a central todo-list on that machine; or one process that asynchronously polls hundreds of connections.

The Google system used asynchronous I/O, maintaining 300 connections open at a time. Mercator used a multi-threaded system, with each thread doing a synchronous HTTP request. At the time, since Java 1.1's HTTP class did not support time-outs, and was otherwise inefficient, the authors wrote their own HTTP module. The UbiCrawler had only 4 threads doing synchronous connections. The crawler at Polytechnic used asynchronous I/O to maintain up to 1000 connection at a time! The Thailand-crawler used up too 300 threads per machine, at which point,

the CPU was the bottleneck (and their system did not do HTML parsing). The Internet Archive crawler maintained 64 connections asynchronously, one for each host currently being crawled.

4.3 DNS

The DNS lookup phase has similar issues as HTTP download: multiple requests should be made in parallel, and should be done asynchronously, or with adequate timeouts. As the authors of Mercator found out, while caching DNS results is moderately effective, the heart of their problem was that the Java interface to DNS was synchronized. Furthermore, the standard UNIX call `gethostbyname()` was as well.²

Their solution (not an uncommon one) was to write their own multi-threaded DNS resolver that could maintain multiple outstanding requests at once, and simply forwarded the request to a real name server.

Note that some computer still needs to do the actual work involved in doing a DNS lookup (which typically requires starting at the .COM root server, and eventually contacting multiple machines, getting to the needed server). Casual internet users typically take this for granted, as their ISP has dedicated machines that provide this service. For a crawler issuing millions of queries, one can't rely on their ISP's machines. The simple solution is to run a caching name-server like BIND on a nearby machine - possibly on the same machine as the other modules.

The Polytechnic crawler used a free package called `adns`, which performs the same request parallelization as Mercator implemented.

4.4 Managing UrlsTodo

To crawl the web with a breadth-first-search, one would simply use an efficient FIFO queue, inserting new urls to the back, and taking urls from the front. However, things are not this simple. The main constraint is politeness: if there are 100 urls from the

²While this has been fixed in later versions of BIND, `gethostbyname()` still has the drawback that calls may block for up to 5 minutes before timing out. If domains like these occur often enough, these 'dead threads' will accumulate and consume resources.

same domain at the head of the queue, we would like to be able to push those aside temporarily and find the next url in line from a different domain.

The straightforward approach, which Mercator uses, is to maintain parallel queues at the head of the big queue, where urls are divided by domain. There are as many small parallel queues as worker threads. In the Polytechnic crawler, there is also a set of small queues at the head of the list - but one for each host-name. A given host-queue is constantly switched between 'ready' and 'politely waiting' states. The Internet Archive crawled domains in a series of 'bundles'. A set of starting urls to a domain is initially given, and one machine exhaustively (but politely) crawls all links within that domain. External links are divided into their bundles, and saved for later. Thus there is no global url FIFO.

It should be mentioned that a global `UrlsTodo` will be very large, especially at the beginning of a crawl, when every url is new. Fortunately, for BFS-like searches, the queue can be stored on disk, with the head and tail in memory, periodically accessing disk.

More generally, one might want a specialized url ordering, like PageRank (that favors highly-linked pages), or topic-driven crawling (that favors links from pages about a certain topic). Strategies like these are only useful when the goal is to obtain a subset of the web, and aims for the best possible subset. For specialized url-orderings the `UrlsTodo` would have to be more sophisticated. Metadata would have to be passed in with each url for information like: "which url contained this link?" and "what was the context (what terms were in the link text)?" Whether a crawling architecture could adapt to a change like this is a testament to its design. Mercator was written with the aim of extensibility, and indeed they mention that one can easily swap in a new implementation of their "Url frontier" module. For example, they implemented a randomized DFS crawler in 360 lines of Java code.

4.5 Updating/Querying UrlsDone

At first glance, this might appear to be a simple task, but in fact turns out to be one of the most difficult parts of a large web crawler. The task

amounts to choosing a good data structure that will efficiently support `.insert(url)` and `boolean .contains(url)`. Supposing one could compress each url to roughly 10 bytes, this would allow roughly 200 million urls inside 2GB of memory. Aiming for 2 billion urls or more will clearly require either splitting this database across multiple machines and/or spilling over to disk. Other than the two operations just mentioned, there are actually few requirements of the `UrlsDone` data structure. This flexibility has enabled a wide range of approaches from previous work. Before getting into them, we have listed some of the key observations about how the data-structure will be used:

- **One `.insert(url)` for each page downloaded.**
- **Ten `.contains(url)` for each page (one for each link in it).**
- **Most links are internal (to the same domain).** That is, there is high domain-locality for `.contains()` operations.
- **Exact operation is not critical.** A failed `.insert()`, or false negative for `.contains()` simply means that the page is mistakenly recrawled. If this occurs infrequently, it may be tolerable. A false positive for `.contains()` means that link will not be followed. Again, if the probability of this is small, it may be acceptable.
- **Immediate response is not needed.** As long as there are other urls todo, the rest of the system can keep busy downloading pages. `UrlsDone` only needs to process (1 insert + 10 contains) for each page downloaded *on average*. Thus, operations can be batched together if this is more convenient.
- **URL string doesn't need to be stored.** Indeed, `.insert(url)` and `.contains(url)` don't necessarily require that the actual url be stored inside `UrlsDone`. For example, one could use a hash-table to store the url *checksums*. If the crawler's purpose is just to crawl each page once, then this could work. However, if enumerating

through the urls is every required (perhaps one wants to recrawl the pages for freshness), then the url string would need to be stored somewhere - either in the `UrlsDone`, or maybe in a different structure (call it `UrlStrings`) on disk that is less-frequently accessed.

The Mercator approach was to store only the url checksums, which was constructed in such a way that the high-order bits correspond to the checksum of the url's hostname. Thus, urls from the same domain are close together. These checksums are sorted on disk, and an LRU cache of 2^{18} entries is kept in memory. They claim a good hit rate, and observe an average of 0.16 disk seeks per operation.

The Internet Archive uses a probabilistic method: the *Bloom Filter*. This in-memory approach begins with a large bit-array initially all set to 0. Inserting a url results in ten different hash functions being computed, and each corresponding bit in the array set to 1. To check if a url has been inserted before, the ten hash-values are computed, and if all entries in the array are set, then `.contains(url)` returns true. Thus, false-positives may occur. The probability of a false-positive starts out very small, but increases as more and more bits are set. Nevertheless, the Bloom Filter is very space efficient. During a crawl, the Internet Archive uses a 32KB Bloom Filter per domain, and uses a 2GB array globally. At the time, when the web was 100 million pages or so, this was more than adequate. Clearly, this method has a practical limit of how many urls it can support. As crawls grown to billions of pages, a 2GB Bloom Filter will eventually have a non-trivial amount of false-positives.

The Polytechnic crawler used a disk/memory structure like Mercator, but was designed to avoid random disk accesses. A Red-Black tree is used to store new urls until a periodic disk-scanning phase is invoked, during which pending insertions are made to disk, and `.contains()` queries are answered. Note that a compressed form of the full url was stored on disk, rather than just checksums.

4.6 Parsing HTML for links

The only issue with HTML parsing is that it should be robust to (not unusual) syntax errors. As it turns out, this is difficult to do fast *and* with high accuracy. Fortunately, the worst-case outcome is the occasional missed link. Google hand-tuned their parser by using a lexical analyzer with a custom stack, which they noted was a time-consuming effort. Aside from thread-switching, this appears to be the only potential CPU bottleneck.

5 Conclusions

A summary and comparison of each system is shown in figure 2.

Numerically, the Thailand crawler is the fastest system, at 618 urls/second. However, their system started with a pre-made list of 400K urls, all of which were from the local .th domain. Their system would never scale to a full internet crawl. Nevertheless, it is interesting to see an example where if the task is simplified to just downloading pages, significant performance results. That is, their performance can be considered a practical upper-bound for systems with similar hardware.

The Mercator project is notably the most successful system, at 600 urls/second covering nearly a billion pages with only 4 machines. Primarily, Mercator is proof that a Java based system can be made to have high performance. However, this required significant rewrites to many of the Java core libraries, and resulted in formal recommendations of changes to the Java specification. Other potential pitfalls including disk-seeks and synchronous, threaded I/O, miraculously did not affect performance. In the end, the thing to be learned most from Mercator is that dedication and hard work pays off.

That exception aside, it appears that using C++ and asynchronous I/O resulted in systems that could maintain thousands of connections without significant CPU consumption. While Bloom Filters are intriguing, they must be built with a particular url limit in mind. Other disk-based approaches, with in-memory caching/buffering are generally success-

ful, assuming random disk accesses are avoided.

DNS lookup and page download are primarily a software difficulty, since parallel non-blocking interfaces to both are generally not available.

The work of a crawler is easily parallelized, and dividing the url-space by domain seems like the best solution. For crawls involving more than a handful of machines, more work needs to be done on what to do when machines enter and leave the cluster ([BCSV02] did address this issue).

6 The Future of Web Crawling

While one can always buy more, bigger, and faster computers, and design smarter software to coordinate them, in the end, network bandwidth will be the limitation: there is only so much that router can funnel into one local area network. And as the size of the internet overseas increases, network bandwidth and latency may become more of an issue. Unfortunately, even if network speeds increase, one would imagine that the amount of data on the internet increases with it (perhaps not textual data, but multimedia, for example).

If network speed indeed stays the bottleneck for web crawlers, then there are two directions of research. The first, which is an active area now, is designing algorithms to learn which pages to refresh when, so that overall database freshness can be maintained with less work. The second, is the potential for wide area distribution, which has some intuitive appeal: a web crawler node in France would be more efficient at crawling French pages. Fortunately, coordination costs between nodes is fairly low (for every 20KB web page downloaded, only a few urls need to be transmitted. However, this assumes that it is okay to leave the documents at the retrieved site. If all the pages must eventually be sent back to a central repository (eg. for indexing), then distribution hasn't solved anything. One help is that HTML can be compressed to 20% size. Furthermore, if only the text is needed (i.e. Javascript and non-essential markup is removed and the resulting text compressed), total compression can be a factor of 10, which might make wide area distribution marginally effective.

	Google	Mercator	Internet Archive	UbiCrawler	Polytechnic	Thailand
UrIsDone		mem hash-table	Bloom Filter	—	mem Red-Black tree	AVL-tree of
Data-structure		disk sorted list	per domain & global		disk sorted list	url suffixes
DNS	cache per node	custom			adms	
Impl. Lang	C++/Python	Java		Java	C++/Python	C++
Special Libs	custom html parse	rewrote Java			adms	
Parallelism	asynch I/O	synchronous	asynch	synch	asynch	synch
(per machine)	300 connects	100's of threads	64 connects	4 threads	1000 connects	300 threads
distribution	4 machines	4 machines		16 machines	3 machines	4 machines
crawl size	24 million pages	891 million	100 million		120 million	400 thousand
crawl rate	48 pages/sec	600 pages/sec	10 pages/sec ?	52 pages/sec	140 pages/sec	618 pages/sec

Figure 2: Comparison of Distributed Web Crawling Systems.

References

- [BCSV02] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. Ubicrawler: A scalable fully distributed web crawler, 2002.
- [BP98] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.
- [Bur97] Mike Burner. Crawling towards eternity: Building an archive of the world wide web. *Web Techniques Magazine*, 1997.
- [CGM00] Junghoo Cho and Hector Garcia-Molina. The evolution of the web and implications for an incremental crawler. In *Proceedings of the Twenty-sixth International Conference on Very Large Databases*, 2000.
- [CGM02] J. Cho and H. Garcia-Molina. Parallel crawlers. In *Proc. of the 11th International World-Wide Web Conference*, 2002.
- [CGMP98] Junghoo Cho, Hector García-Molina, and Lawrence Page. Efficient crawling through URL ordering. *Computer Networks and ISDN Systems*, 30(1–7):161–172, 1998.
- [EMT01] Jenny Edwards, Kevin S. McCurley, and John A. Tomlin. An adaptive model for optimizing performance of an incremental web crawler. In *World Wide Web*, pages 106–113, 2001.
- [HN99] Allan Heydon and Marc Najork. Mercator: A scalable, extensible web crawler. *World Wide Web*, 2(4):219–229, 1999.
- [HRGMP00] Jun Hirai, Sriram Raghavan, Hector Garcia-Molina, and Andreas Paepcke. Web-Base: a repository of Web pages. *Computer Networks (Amsterdam, Netherlands: 1999)*, 33(1–6):277–293, 2000.
- [KaS] Kasom Koht-arsa and Surasak Sangunpong. High performance large scale web spider architecture.
- [NH] M. Najork and A. Heydon. On high-performance web crawling.
- [NW01] Marc Najork and Janet L. Wiener. Breadth-First Crawling Yields High-Quality Pages. In *Proceedings of the 10th International World Wide Web Conference*, pages 114–118, Hong Kong, May 2001. Elsevier Science.
- [SS02] Vladislav Shkapenyuk and Torsten Suel. Design and implementation of a high-performance distributed web crawler. In *ICDE*, 2002.
- [ZYD] Demetrios Zeinalipour-Yazti and Marios Dikaiakos. Design and implementation of a distributed crawler and filtering processor.