Building a Peer-to-Peer, domain specific web crawler

Tushar Bansal
tusharb @ cc.gatech.edu

Ling Liu
lingliu @ cc.gatech.edu

ABSTRACT
The introduction of a crawler in mid 90s opened the floodgates for research in various application domains. Many attempts to create an ideal crawler failed due to the explosive nature of the web. In this paper, we describe the building blocks of PeerCrawl - a Peer-to-Peer web crawler. This crawler can be used for generic crawling, is easily scalable and can be implemented on a grid of day-to-day use computers. Also, we demonstrate and implement a novel scheme for coordinating peers to follow a focused crawler. We cover the issues faced during the building of this crawler and decisions taken to overcome the same.

Keywords
Web Crawler/Spider, Peer-to-Peer, Bloom Filter, Threads

1. INTRODUCTION
The web has expanded beyond its expectations to give rise to a chaotic monster of information in varied forms of media. Many systems heavily depend on retrieval of such information to various uses ranging from critical medical prescriptions to simply fun facts. A common thread amongst all these applications is a web crawler whose job is to gather data connected via hyperlinks. Since the anatomy of a search engine by Google[1], many attempts have been made to explain the composition and working of a crawler. As most of the centralized web crawlers suffer from obvious disadvantages, fine tuning their performance by adding more correlated components is really a short term fix.

The advent of problems in a generic crawler has resulted in research on various types of web-crawlers. Sphinx[2] was one the first attempts towards a domain-specific web crawler. The definition of a domain has ranged from a web-domain, topic-specific focused crawling[3] to a type of document media (images, pdf, etc). Also there has been a lot of work on incorporating techniques from Artificial Intelligence[4,5] in order to improve quality of crawling. Finally, the introduction of distributed systems has produced a new breed of high performance crawlers. Vladislav, et al[6] and Bold, et al[8] give a detailed description of a distributed web crawler, while Cho[7] gives an overview of parallel crawlers.

As web crawlers have been the point of interest for so many years, there is a buzz to generalize the architecture of a crawler. Mercator[10] did an excellent job in highlighting the problems faced during construction of a crawler while proposing a generic structure for a crawler. With the aid of Mercator[10] and other systems like [6,11], not only we have various models to build a crawler, but also the common problems faced.

This paper focuses on using the advantages of Peer-to-Peer(P2P) systems in web crawling. The functionality of PeerCrawl can be extended from a scalable, generic-crawler to a web-domain specific crawler. It can thus be used to study the structure of a particular website and further be used to create source-specific page rank[9] for efficient crawling.

P2P systems have demonstrated their scalability and versatile usage through numerous applications. The main advantage of P2P systems is that they can be implemented on a grid of normal day-to-day use computers. This proves as a lucrative idea for an exhaustive task like web crawling.

The rest of the paper is divided into the following sections. In Section 2 we will look at the system architecture for one peer. In Section 3 we will study the coordination of various peers. In Section 4 we will try to point different issues involved with building a crawler. Finally we discuss implementation details & experimental results in Section 5.

2. SYSTEM ARCHITECTURE
We design our crawler based on Mercator[10], which gives a good overview of the essential components of a crawler. It also justifies the presence of a component with different issues faced while creating a crawler, along with some performance measures. However, the main difference in this crawler is the presence of a network layer which is used to communicate between the different peers. Let us look at the architecture of PeerCrawl in more detail.

Each peer in PeerCrawl is broken into many threads of control, which perform various tasks ranging from fetching a document for a web server to maintaining bookkeeping information. All these threads are autonomous, but they could share some data structures between other threads, thus leading to various synchronization issues. Note that all these threads and data structures are for one peer only.

2.1 Data Structures
Firstly, let us discuss the different data structures that different threads could operate on.

- **CrawlJobs**: List of urls which are fetched and processed by different threads
Figure 1. Workflow of single node in PeerCrawl

- **FetchBuffer**: List of documents (web pages/pdf/doc/etc..) that are processed and cached by some threads
- **UrlInfo**: Map of Bloom Filters for every domain, used to detect duplicate urls that are already crawled

### 2.2 Threads

Now, let us look at the different types of threads that constitute the working engine of PeerCrawl.

- **Fetch_Thread**: The main purpose of this thread is to fetch a document from a web server. It takes a url from *CrawlJobs* and gets a document using a standard HTTP Connection.

- **Process_Thread**: This thread scans a document from *FetchBuffer* for urls using standard Pattern Recognition techniques for URLs. They also keep a track of duplicate urls using bloom filters stored in *UrlInfo*.

- **Caching_Thread**: Caches documents from *FetchBuffer* to secondary storage for later retrieval during user queries. The cached documents can be used to create an internet archive for specific domains.

- **Statistics_Thread**: Maintains all the bookkeeping information for ever peer. These include no of urls crawled, added and processed per second, various HTTP connection statistics along with the dropped & duplicated urls.

- **Backup_Thread**: Used to periodically backup critical data structures for recovery purpose.

- **Network_Connection_Thread**: This thread uses the P2P network layer to detect any peers entering or leaving the network.

- **Dispatch_Thread**: A Peer broadcasts any url that doesn’t belong to its domain of crawl. The domain of a peer is dynamically updated depending on the number of peers in a network at any instance.

More information regarding the interaction between different peers is detailed in Section 3.

### 2.3 Workflow

After having an idea of the different types of data structures and threads, let us walk through the life-cycle of a peer in PeerCrawl. It starts with a seed list of urls which are typically diverse urls for giving a better start range for the crawler.

0) A seed list of urls is added to *CrawlJobs*

1) **Fetch_Thread** picks up a block of urls

2) **Fetch_Thread** gets a url from web servers. It could use a local DNS cache to fasten connection setup to web server
3) If there are no HTTP connection errors or socket timeouts, Fetch_Thread puts document into FetchBuffer for processing.

4) Process_Thread scans document for links to other urls. Urls are normalized and cleaned before passing on to the next filter. Also we limit the urls to a particular depth, so that we don’t run into unnecessary links that loop (e.g page relatively referring to itself).

5) If the document is fully processed, Caching_Thread buffers the document to secondary storage.


7) If a url is not mapped to a peer then it is broadcasted to other peers. Otherwise it is added to CrawlJobs to complete the cycle.

The Statistics_Thread, Network_Connection_Thread & Backup_Thread keep running in the background with different time intervals.

3. PEER COORDINATION

So far we have seen the working of a single node in PeerCrawl along with its data structures and thread types. But the difference in PeerCrawl from its counterparts is the notion of a P2P network. We use an open-source Gnutella[13] as the underlying P2P network. Previous work in this project[14] describes the working of the P2P layer in more detail. In this paper, we would concentrate on the use of this layer to establish various forms of Peer coordination.

During the first few iterations of the crawler, the primary motive was to crawl the entire web. The peers used a URL Distribution function as described in DSphere[15] which would dynamically decide the floor and ceiling of the crawl domain of a Peer. This was done by equally dividing the total range of IP Addresses amongst the peers. Although PeerCrawl was feasible for crawling the entire world wide web, the ingenuity lies in coordinating the peers to crawl a web-domain or a set of domains. We propose atleast two variations of doing this task where we assume that the user will give a set of domains to be crawled as a Task-List (different from seed list).

One possible way of coordinating the peers is by assigning each peer with one domain from the Task-List. This would result in ideal performance if:

No of. Peers <= | Task-List |

If the no of peers are greater than the task-list, then some peers will sit idle. There is very less interaction between the peers as each of them crawl non-overlapping domains. The root node can be made as a center for getting uncrawled domains.

Typically in real world scenarios, the Task-List could be small and the number of possible peers could be large. The best way to use the scalable nature of the P2P network is to convert the Task-List into a continuous domain, thus allowing mapping similar to one described in DSphere[15]. This will even ensure that domains with uneven sizes get divided uniformly. However, this will increase the communication between peers along with the rise of duplicate urls.

There could be a hybrid version incorporating both the schemes mentioned above. However, the performance of these schemes could be largely influenced by the nature of the domain i.e. the size, connectivity, depth, etc. This field of research is rather unexplored and could probe novel schemes for achieving maximum performance gain.

4. ISSUES WITH CRAWLING

One of the standard problems placed with any data-intensive system is managing overflows of data queues. This coupled with thread scheduling makes it a very interesting problem from the systems point of view. In any system, as the number of threads increase, the synchronization overhead becomes more prominent. Thus, there is always a classic tradeoff between synchronizing data structures and keeping a local copy of the same. We will try to justify a blend of both schemes.

4.1 Buffer Management

Synchronization is needed for the CrawlJobs and FetchBuffer data structures, which traditionally operate as FIFO queues. Later on, CrawlJobs could operate as priority queues with enhanced schemes to determine the importance of urls (e.g. page ranking). In Section 5, we will examine the limits for each of these data structures with respect to various varying factors. Note that these data structures are also synchronized by Statistics_Thread and Backup_Thread.

The Process_Thread adds a block of urls to the CrawlJobs queue which the Fetch_Thread picks up and gets the document from the web. Urls are added in a block due to urls found in a document. Thus, we allow the Fetch_Thread makes a local copy of the queue, which reduces synchronization overhead from constantly spinning on CrawlJobs. There is a limit (empirically determined) for CrawlJobs which results in a spawn of a new Fetch_Thread to handle the overflowing urls. Other design techniques include that of local queues per process and fetch thread in Mercator[10].

The FetchBuffer works quite differently from CrawlJobs. The threads synchronizing over FetchBuffer are: Fetch_Thread which adds documents, Process_Thread which gets and processes documents and Caching_Thread whose job is to buffer the processed pages to secondary storage. Since there is fixed ordering amongst these
4.2 Thread Scheduling

Many crawlers follow a custom scheduling for various threads in their system to improve performance. We argue that if the system is implemented in JAVA, the best way to improve performance is to leave thread scheduling to JAVA Virtual Machine (JVM). If we want to explicitly control the timings of some threads, (like Backup_Thread, Statistics_Thread in this system), we can use thread routines provided by JAVA. However, we should decide on the number of threads running on the Peer. We have tried various schemes and found the best performance is given by taking pairs of Fetch_Thread and Process_Thread apart from other combinations.

4.3 DNS Lookup

According to Mercator[10], the maximum time lost during the life-cycle of a crawler is that during DNS Lookup. Although we have not done stress testing on this issue, we say that JAVA caches the DNS lookups internally which removes the need for a separate user level data structure for the same.

4.4 URL Starvation

The universal problem with all crawlers is that it would run out of urls if left to venture with remote urls. The solutions floating around include improving the quality of seed list (which fails as it is very difficult to find a good quality seed list) to finding similar document using meta-crawl techniques[16]. We adopt tunneling proposed by Bergmark[17] as an approach to venture out of the web-domain to find links pointing to pages within the domain. We control this by maintaining a hop count which can be empirically set.

4.5 Other Issues

We follow the policies that a traditional web crawler should follow including Politeness Policy and Re-Visit Policy apart from others. We also follow the Robots Exclusion standard[12] that disallows the crawling of some sub-domains for different types of user agents.

5. EXPERIMENTS AND RESULTS

In this paper we concentrate on performance issues of one peer in PeerCrawl. Since we have a lot of varying parameters, we can setup some interesting experiments to study the crawler. We run all our experiments on the domain of gatech.edu. First, let us look at some domain statistics from well known crawlers about gatech.edu.

<table>
<thead>
<tr>
<th>Search Engine</th>
<th>Size of gatech.edu</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google</td>
<td>1,240,000</td>
</tr>
<tr>
<td>Yahoo! Search</td>
<td>1,445,287</td>
</tr>
<tr>
<td>MSN Search</td>
<td>576,358</td>
</tr>
<tr>
<td>Ask.com</td>
<td>1,485,000</td>
</tr>
</tbody>
</table>

Table 1. No of urls according to various search engines

If we look at the results according to Figure 2, the P2P crawler performs well, considering this is just the number of URLs crawled by a singleton node in the P2P network, and by starting with a single seed URL. A depth is defined as a sub-domain of a domain. We also observe that the number of urls increases exponentially with depth in the initial stages and then settles to a linear graph. This is true as the number of urls in the first few sub-domains is much more than in the deeper ones.

![Figure 2. Crawled vs Depth of domain](image)

The result according to Figure 3 gives us an example of the impact of the number of threads on the performance of any node.

![Figure 3. Time to crawl vs No of threads](image)

In the domain of gatech.edu, we have a lot of varying parameters, we can setup some interesting experiments to study the crawler. We run all our experiments on the domain of gatech.edu. First, let us look at some domain statistics from well known crawlers about gatech.edu.
As expected, the time taken decreases exponentially, until it reaches a stable point where it doesn’t go any faster. The deviations in the graph indicate the tradeoff made in the handling of threads. Obviously the point of stabilization shifts more to the right (i.e. more threads are used to reach the minimum time to crawl) as we increase our crawl limit. Also, after a high number of threads, the time to crawl will increase due to the overhead of handling the excess threads.

In the end let us examine the results for the size of different data buffers that the threads synchronize on. As expected, the size of CrawlJobs decreases as more Fetch_Thread are introduced. However, the size of FetchBuffer increases as the Fetch_Thread is faster than Process_Thread.

![Figure 3. Size of FetchBuffer & CrawlJobs vs No of Threads](image)

6. CONCLUSION

We have demonstrated the working of a node in PeerCrawl along with the factors affecting its performance. We can expect better performance and more extensibility to the crawler as we use a number of peers in coordination. Although, we have to still fine tune the crawler, we can postulate its usage in many domains apart from the traditional indexing and searching.

7. REFERENCES


