Enhancement to PeerCrawl: A Decentralized P2P Architecture for Web Crawling

Mahesh Palekar, Joseph Patrao.

Abstract:
Search Engines like Google have become an Integral part of our life. Search Engine usually consists of 3 major parts - crawling, indexing and searching. A highly efficient web crawler is needed to download billions of web pages to index. Most of the current commercial search engines use a central server model for crawling. Major problems with centralized systems are single point of failure, expensive systems and administrative and troubleshooting challenges. Hence a decentralized crawling architecture can overcome these difficulties. This project involved making enhancements to “PeerCrawl” a decentralized crawling architecture. Enhancements involved adding GUI, bounding buffers. We have achieved a crawl rate of 2500 urls per minute while running our crawler with four peers while running eight peers we achieve a crawl rate of 3600 urls per minute.

1. Introduction:
Search Engines like Google have become an Integral part of our life. Search Engine usually consists of 3 major parts - crawling, indexing and searching. Crawling involves traversing the WWW by following the hyperlinks and downloading the web content to be cached locally. Indexing organizes web content for efficient search. Searching involves using indexed web content for executing user queries to return ranked results. A highly efficient web crawler is needed to download billions of web pages to index. Most of the current commercial search engines use a central server model for crawling. Central server determines which URL’s to crawl and which URL’s to dump or store by checking for duplicate URL’s as well as determining mirror sites. Along with central server being single point of failure, server needs to have very large amount of resources and is usually very expensive. Mercator [7] used 2GB of RAM and 118 GB of local disk. Google also has a very large and expensive system. Another problem with centralized systems is congestion of link from the crawling machine to the central server. Also need experienced administrators are needed to manage and troubleshoot these systems. In order to address the shortcomings of centralized crawling architecture, we propose to build a decentralized peer – to – peer architecture for web crawling. The decentralized crawler exploits excess bandwidth and computing resources of the clients. The crawler clients run on desktop PCs utilizing free CPU cycles. Each client is dynamically assigned a set of ip addresses to crawl in a distributed fashion.

The major design issues involved in building a decentralized web crawler are:

- Scalability: The overall throughput of the crawler should increase as number of peers increase. The throughput can be maximized by two techniques:
  - By distributing load equally across the nodes
  - Reducing the communication overhead associated with network formation, maintenance and communication between peers.
• Fault Tolerance: Crawler should not crash on failure of a single peer. On failure of a peer, dynamic reallocation of URL’s must be done across other peers. Crawler should be able to identify crawl traps as well as be tolerant to external server failures.
• Efficiency: Optimal crawler architecture improves efficiency of the system.
• Fully Decentralized: Peers should perform their jobs independently and communicate required data. This will prevent link congestion to a central server.
• Portable: The crawler should be able to be configured to run on any kind of overlay network by just replacing the overlay network.

The main objective of the project is not proposing any new concept but building on the previously done research to implement a decentralized p2p crawler. The paper discusses the architecture of a prototype system called “PeerCrawl” developed at Georgia Tech. PeerCrawl is a decentralized crawler running on top of Gnutella. Section 2 discusses related work. Section 3 describes the Crawler Architecture along with the architecture of a single Peer. Section 4 describes features and shortcomings of the current implementation. In Section 5, we present a detailed analysis of our experimental results. Section 6 presents the motivation and requirements for a User interface for such a system and finally section 7 summarizes our work.

2. Related Work
Most of the crawlers have centralized architecture. Google[5], Mercator[7] have single machine architecture. They deploy a central coordinator that dispatches URL’s. [3, 4] presents a design of high performance parallel crawlers based on hash based schemes. These machines suffer from drawbacks of centralized architecture discussed above. [1, 8] have developed decentralized crawlers on top of DHT based protocols. In DHT based protocols, there is no control over which peer crawls which URL. It is dependent on the hash value. So a peer in U.S.A may be assigned an ip addresses in Japan and vice versa. It has been shown in [1] geographical proximity affects the crawler’s performance. Crawling a web page in U.S.A is faster than crawling a web page in Japan from a machine at Georgia Tech, Atlanta, U.S.A. Another issue with DHT was that the protocol is pretty complicated and that makes the implementation flaky. Gnutella protocol is much simpler and is widely used for file sharing. Hence the implementation of Gnutella is more stable. Gnutella protocol in itself does not have an inbuilt URL distribution mechanism. Hence it is possible to develop a URL distribution function that assigns nodes only URLs those are close to it. PeerCrawl [2] developed at Georgia Tech uses Gnutella as underlying network layer.

3. System Architecture
The system consists of 3 major components.

- **Overlay Network Layer** – This layer is responsible for formation and maintenance of p2p network, and communication between peers. Overlay networks can be of two major types – Unstructured networks like Gnutella, or structured networks like Chord. PeerCrawl uses flat Gnutella architecture.

- **URL and Content Distribution function** – This function determines which client is responsible for which URL. Each client has a copy of this function. URL & Content range associated with a client may change due to joining/leaving of nodes in the network. This block is very essential for load balancing and scalability. A good distribution function will distribute URL’s as well as content equally among all peers. This function should make optimum use of the underlying overlay network in order to provide load balancing and scalability resulting in improved efficiency. It has been shown that it takes less time to crawl web pages geographically close to peer than pages far away from the peer [1]. A good mapping function should take into account geographical location and proximity of nodes.

- **Crawler** - This block downloads and extracts web pages. It sends and receives data from other peers using the underlying overlay network. A single peer consists of following blocks:

1. **URL Preprocessor**: Whenever a node receives a URL for processing from another node. It adds the URL to one of the crawl job queues. URL processor has following blocks:

   - **Domain Classifier**: This block determines to which crawl job queue a request should be added. All URL’s from the same
domain are added to the same crawl job queue. This is not yet implemented.

- **Ranker** – Current version of PeerCrawl crashes after 30 minutes because rate of Input is much faster than the rate of output. It is impossible to crawl all URL’s. Hence it makes sense crawling only important URL’s. Ranker block calculates the importance of the URL by calculating page Rank [6]. All URL’s below certain ranking threshold should be dropped. Currently ranker block is not implemented in PeerCrawl.

- **Rate Throttling**: This block prevents excessive request to the same domain. In order to prevent overloading of the web server, rate at which URLs are added the crawl job queue is controlled. This is not implemented.

- **URL Duplicate Detector**: This block checks whether URL has already been processed by accessing the Seen URL data structure.

2. **Crawl job queue**: Holds URL’s to be crawled. All URL’s from the same server are put in the same queue.

3. **Downloader**: Downloads web pages from the server. There is one downloader thread for each crawl job queue. It keeps connection open to a server. This eliminates TCP connection establishment overhead. Downloader also needs to check for Robot Exclusion pages and not crawl them.
4. **Extractor**: Extracts URLs from the pages and passes it on to the URL Range validator. Extractor is multithreaded and processes different pages simultaneously. Downloader also needs to check for Robot Exclusion pages and not crawl them.

5. **URL Range Validator**: URL range validator checks whether URL lies in the URL range of the node. If it lies within URL range of the node then it sends the URL to the URL preprocessor. Else sends it out on the network. In case of Gnutella, URL is broadcasted over the network.

6. **Content Range Validator**: Checks whether content lies in the range of client. This is not yet implemented.
7. **Content Processor**: It checks page for duplication from Seen Content data. If page is not already processed, then the page is added to the content seen data store. This is not present in the current version of the software.

8. **Local page duplication interface**: Multiple threads pick up a URL from the processing jobs queue. They check whether the peer is responsible for the content by calling Content Range Validator. If the peer is responsible for the content then it calls content processor. Else sends a content query on the network. If the content is already processed, the page is dropped else the page is added to the pending queue. This block is not implemented.

9. **Remote page duplication interface**: This block provides an external interface for content duplication queries. This block is also multithreaded. It calls content processor and output of the content processor is sent back to the requester. This block is not yet implemented.

10. **Processing Jobs Queue**: This queue stores web pages which have been downloaded and to be processed. This is a FIFO queue. Current implementation does not have a processing jobs queue.

11. **Pending job queue**: This queue feeds the extractor.

12. **Seen Content Data Store**: It stores web pages that are already processed.

13. **Seen URL Data store**: It stores URL’s that are already processed.

4. **Implementation**:

Phex client (hereafter referred to as Phex layer) constitutes the network layer of PeerCrawl and encompasses the API’s for Gnutella protocol. The crawler component is built upon the Phex layer and makes calls to the routines in this layer. The crawler uses the following data structures:

- Crawljob Hashtable – data structure where domains are hashed.
- Crawljobs – queue containing list of URLs to be crawled.
- Pending – queue-containing list of URLs extracted from the downloaded pages.
- Batch – queue containing list of URLs to be broadcasted.
Each domain encountered is hashed in the Crawljob Hashtable. Each hashed entry has an associated crawljob queue. The downloader and extractor thread are combined into a single thread named “fetch”. Multiple fetch threads service the crawljob hashtable. A thread finds the first available entry in the hashtable and grabs all the URLs in the queue associated with that particular entry. Thus we seek to maximize the number of URLs crawled per HTTP connection.

The “fetch thread” picks up URLs from the crawljobs queue and adds the URLs extracted in the process to the pending queue. URLs from this queue are processed and added to the crawljobs queue if they are to be crawled by the current peer or to the batch queue if they are to be broadcasted to other peers. Each URL from the batch queue is packaged in the payload of the query message and broadcasted.

URL distribution function determines the crawling a range of URL domains. The IP address of peers and domain of the URLs are hashed to the same value space. The domain range of a peer \( p_i \) is computed as:

\[
\text{Range}(p_i) = h(p_i) - 2^x \text{ to } h(p_i) + 2^x
\]

where,

- \( h \) is the MD5 hash function
- \( x \) is chosen so that the hash value space is distributed among the peers. \( x \) is dependent on number of peers in the network. In current implementation is dependent on number of neighbors of the peer. Whenever a node enters/leaves the network, ip range of all peers change.

A URL \( u_i \) is crawled by peer \( p_i \) only if domain \( d_i \) of \( u_i \) is in the range of \( p_i \) as computed above. If it’s not in range, the URL is broadcasted by the peer. The receiving peers drop the URL if it’s not in range.

The rate at which URLs are extracted is faster than the rate at which URLs are downloaded. On fetching one URL, most of the times, more than one URL is added to the queue. Hence the system is inherently unstable. In order to stabilize the system and avoid the crashing of the crawler due to memory overflows, we seek to bind the buffers. We set a max limit and a min limit on the crawl job queue as well as Pending queue. As
the max limit is reached, no more URLs are added to that queue. We also set a max limit for the number of Domains in the Crawl Job hash table.

5. Evaluation
The crawler was evaluated on the machines in the CoC undergrad lab. The crawler was run for a time period of four hours on 2.99 GHz machines - each with 512MB RAM. The max-min bounds on the Crawl job queue, Pending queue, and number of domains were max-400, min-200; max-400,min-250; 50 respectively.

Figure 3
Figure 4 shows the plot of the number of URLs Crawled per minute versus the number of threads. We have a relatively steady speedup as we increase the number of threads to 10. After that we observe the speedup curve flattening out a bit.
Figure 4 shows the speedup curve of the number of URLs crawled per minute versus the number of peers. The crawler was run with 1, 2, 4, and 8 peers respectively. We see a big increase (theoretically it should be exponential) as we increase the number of peers. The huge jump from 2 peers to 4 peers is probably a statistical error caused by less number of runs.

Figure 5 shows the speedup curve of the number of URLs crawled per minute versus the number of peers. The crawler was run with 1, 2, 4, and 8 peers respectively. We see a big increase (theoretically it should be exponential) as we increase the number of peers. The huge jump from 2 peers to 4 peers is probably a statistical error caused by less number of runs.
Figure 6 shows the vast difference in crawler efficiency when the limits of the buffers are unbound as opposed to when they are bound. The above reading were taken for Crawl Job Queue Size = max 400, min 250; Pending queue size = max 400, min 200; Number of Domains = 50. The crawler was run for hours and acquired 99% CPU utilization.

From Figure 6 & 7 it is clearly evident that the bounds we placed on the buffers reduced its efficiency manifold and that these artificial bounds must be tuned to achieve greater efficiency in terms of the number of URLs crawled per minute.

Figure 7 shows us that with set bounds we the crawler runs for a much longer time (4 hours) as opposed to the 30 minutes when it crashes with unbounded buffers.

6. GUI
A Graphical User Interface was developed which displays the statistics of the crawler in real time. It also allows for simple user queries. The user could check whether a particular URL has been crawled yet. Also it allows for checking the number of times a particular domain has been crawled.
The GUI could also work alone simply outputting data written in the PeerCrawl log files.
Figure 8 and 9 depict the number of URLs fetched – crawled and added to the crawl job queue respectively. Figure 10 shows the content crawled so far. As rightly suspected, most of the data on the internet is text/html.

7. Summary
The decentralized p2p crawler “PeerCrawl” was enhanced by adding in GUI as well as bounding the queues. GUI developed is very intuitive and helps to understand the performance of the crawler. Initially the crawler used to crash in 30 min. But on bounding queues, it can run for over 4 hours. Evaluation of PeerCrawl offers the first set of measurements of the architecture. Our crawler with 8 peers crawls at the rate of 3600 urls/min. At this rate, we could crawl a billion web pages in approximately six months, two days and 22 hours.

The current implementation of PeerCrawl has a lot of shortcomings.

- **Flat architecture**: The current prototype has flat network architecture. All the peers are at the same level. One of the major problems with the flat architecture is flooding of data by all peers. The bandwidth constraint limits the scalability of this architecture. This can be overcome by using supernode architecture in which only super nodes flood data while the leaf peers only talk to the “supernode”. This structure can be used for controlled flooding and load balancing. A “supernoder” can manage the crawl tasks for its leaves so that one particular peer is not overloaded when there are other idle peers or peers with lesser load.

- **Naïve URL distribution function**: The current URL distribution function is really naive. It needs to be modified in order to improve load balancing and
scalability. It should also exploit the geographical proximity of the peer for a particular domain while assigning URLs.

- **Memory Leak:** In the current implementation, we have observed a memory leak. This could be because of a variety of reasons namely unbound buffers. Even by setting bounds on all buffers, the memory leak persists. We can confidently speculate that this maybe due to the way threads are implemented. A single thread once created continues to run for the entire duration of the crawl viz. we cannot say with confidence that the JVM performs thread clean up effectively. To avoid this, we propose the usage of a thread pool and programmer enabled creation and destruction of threads. Thus thread creation and destruction (clean up) can be performed explicitly by the programmer. We think that short threads and user control over threads can solve the problem.

- **Bounding Buffer:** Currently, after the limit is reached new URLs or domains to be added are dropped arbitrarily. This may lead to crawling of lot of unimportant URLs and dropping of important URLs. Dropping strategy can be improved by calculating rank of a URL and then dropping URLs having rank below certain threshold. This will lead to crawling of more important URLs which would improve the significance of the output of the crawler for the applications.

- **Hard Coded bounding limits:** The current configuration of PeerCrawl has arbitrary limits hard coded into it. As evident from Figure 7, the PeerCrawl can be further optimized by tuning the limits (max and min) of the Crawl Job queue, URL store, CrawlJob Hashtable and Pending Queue.

8. References:

9. Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans

