CS8803 Project Proposal

Enhancement to PeerCrawl: A Decentralized P2P Architecture for Web Crawling

Mahesh Palekar, Joseph Patrao.

Motivation and Objective:

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. These systems need experienced administrators to manage the system. In order to address the shortcomings of centralized crawling architecture, we propose to build a decentralized peer – to – peer architecture for web crawling. The distributed crawler exploits excess bandwidth and computing resources of clients. A distributed crawler has two major parts – crawler and network layer. Crawler runs on top of network layer. Crawler downloads web pages and extracts URL's. The network layer is responsible for formation of overlay networks, communication between peers, routing and URL distribution. Protocols like Gnutella [10], Kazaa, CHORD [9] etc are implemented in the Network layer. 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 and 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 purpose of the project is not proposing any new concept but building on the previously done research to implement a decentralized p2p crawler. Currently the PeerCrawl system runs for maximum 30 min with 4 nodes and downloads 40 URL's/sec. The project aims at improving the efficiency of the crawler and scaling it to run on more number of nodes.

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 distributed crawler 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. The performance of both these protocols was poor. [8] lasted for maximum 15 min while [1] just crawled 18 URL's /sec. It has been shown that performance of Gnutella is better than DHT based schemes. Hence using Gnutella protocol in the network layer may provide better performance. PeerCrawl [2] developed at Georgia Tech uses Gnutella as underlying network layer. PeerCrawl currently has a throughput of just 40 URL's/sec and crashes after 30 min because of memory issues. The implementation of PeerCrawl is naïve and we plan to enhance the features of PeerCrawl in order to improve its efficiency and scalability.

Proposed Work:

We plan to extend current PeerCrawl implementation.

System Overview:

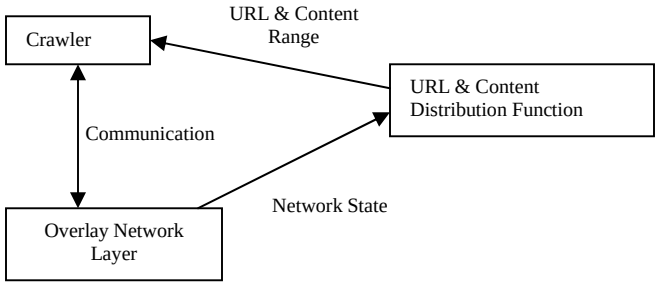

Fig 1: System Architecture

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 based on Gnutella, Kazaa or structured networks based on Chord. Gnutella in its original state is not scalable because of broadcast. Hence super node architecture can be used to improve performance. We plan to use Gnutella as overlay network layer. PeerCrawl currently uses phex gnutella client. The Phex Client has support for normal as well as supernode architecture. We will use the phex client initially with flat architecture. Then if we have time we will port our crawler on top of supernode architecture. If time permits we can try our crawler on other Gnutella clients like limewire etc.
- 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. In our project we will have a static function that determines the URL as well as Content range of the client and won't consider entry/exit of nodes. Range assignment function will be a hash function.
- Crawler: This block downloads and extracts web pages. It sends and receives data from other peers using the underlying overlay network. Crawler architecture is described in the next section.

The compartmentalized architecture offers numerous advantages. It helps to make the system portable. There is a high degree of functional independence between the 3 components of the system. Each of the component is pluggable, thus we can run the crawler on top of different underlying overlay networks or URL content distribution function with minimal regression factor. URL content distribution function should be optimized to make use of the underlying architecture. Combining of the three components will be done using a config file that contains definition of communication interfaces of the overlay network and also other required parameters.

Crawler architecture:

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.

- Ranker – Current version of PeerCrawl crashes after 30 min 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 pageRank[6]. All URL's below certain ranking threshold should be dropped. We are not going to implement Ranker block. But we will implement an overflow controller that drops URLs after the crawl job queue overflows.
- 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 to the crawl job queue is controlled.
- URL Duplicate Detector: This block checks whether URL is already been processed by accessing the Seen URL data structure. If URL is not already processed then the URL is added to the URL seen data structure.

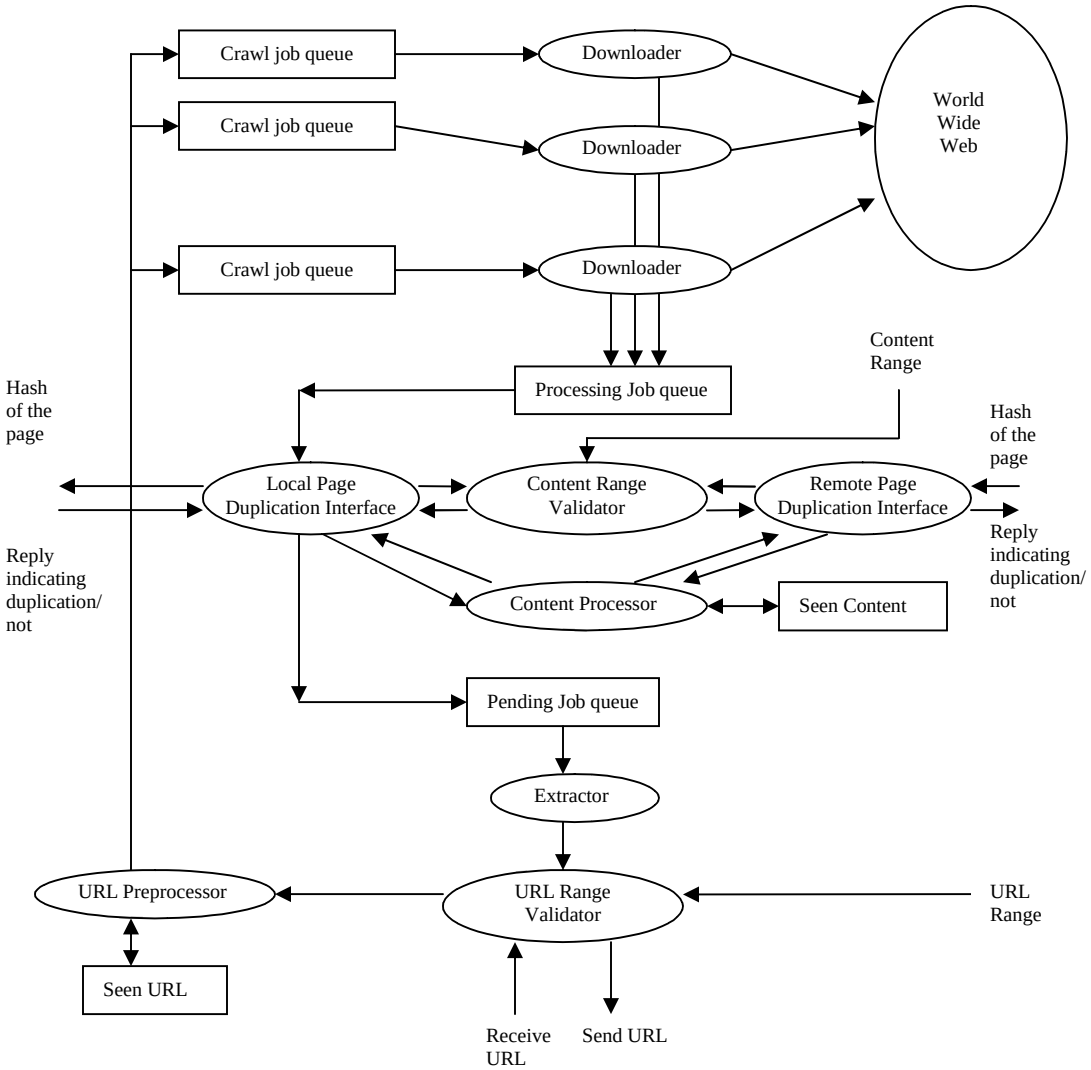

Fig 2: Crawler Architecture

2. Crawl job queue: Holds URL's to be crawled. All URL's from the same server are put in the same queue. Current implementation has only a single queue. We plan to implement multiple crawl job queues.

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. Multiple downloader threads are already implemented in the peerCrawl. PeerCrawl is implemented in Java. Java does not support multiple outstanding DNS queries. This slows down the fetching of webpages. In order to improve this, customized DNS resolver needs to be written. Downloader also needs to check for Robot Exclusion pages and not crawl them. Robot Exclusion feature is not present in the current version of peerCrawl system. For this semesters work, Robot exclusion and DNS resolver have low priority.

4. Extractor: Extracts URLs from the pages and passes it on to the URL Range validator. Extractor is multithreaded and processes different pages simultaneously.

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.

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.

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.

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.

10. Processing Jobs Queue: This queue stores web pages which have been downloaded and to be processed. This is a FIFO queue.

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

12. Seen Content Data Store: It stores web pages that are already processed. This module will be grabbed from the Apoidea[1].

13. Seen URL Data store: It stores URL's that are already processed. This module will also be grabbed from the Apoidea[1].

Evaluation and Testing:

The crawler will be evaluated by running it on 16 machines in the CoC. Performance will be compared with current PeerCrawl[2] implementation and Apoidea[1]. Following graphs will be used to evaluate the performance

- # of URL's crawled vs time.
- # of URL's crawled/sec/peer vs # of peers: This graph will indicate the scalability of the crawler.
- # of URL's crawled/sec vs different CPU loads: Distributed web crawler is assumed to run on the spare CPU processing power at client machines. Hence it should not burden the client machine too much.
- # of bytes downloaded/sec vs # of peers : This graph is another way of demonstrating scalability.

Performance evaluation will also be done to tune the crawler i.e.

- Throughput of a peer vs # of crawled job queues: This will help in determining the optimal # of crawl job queues.

Project Plan:

Hardware Resources:

Atleast 16 Windows machines.

Plan of Action

Sr. No	Due Date	Milestones
1	Feb 30	• Installation of Peer Crawl • Literature Survey on Gnutella Protocol. • Understanding of Source Code of PeerCrawl.
2.	March 15	• Execution of PeerCrawl on 16 clients • Performance Evaluation of Current versions of PeerCrawl.
3.	March 30	• Implementation o URL Seen Test. o Content Seen Test.
4.	April 7	• Implementation o Multiple Crawl Job Queues.
5.	April 15	• Implementation o URL preprocessor. o Porting on other overlay network if time permits.
6.	April 22	• Performance Evaluation of modified peercrawl.
7.	April 29	• Final Project Report.

Reference:

1. A. Singh, M. Srivatsa, L. Liu and T. Miller, “Apoidea: A Decentralized Peer-to-Peer Architecture for Crawling the World Wide Web”. In the proceedings of the *SIGIR workshop on distributed information retrieval*, August 2003.
2. Padliya Vaibhav, “PeerCrawl”. Masters Project.
3. V. Shkapenyuk and T. Suel. “Design and implementation of a high-performance distributed web crawler”. In *ICDE*, 2002.
4. J. Cho and H. Garcia-Molina. “Parallel crawlers”. In *Proc. Of the 11th International World–Wide Web Conference*, 2002.
5. S. Brin and L. Page. “The anatomy of a large-scale hypertextual Web search engine”. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.
6. Junghoo Cho, Hector Garcia-Molina, and Lawrence Page, “Efficient Crawling through URL Ordering”. *Proceedings of the 7th International World Wide Web Conference*, pages 161-172, April 1998
7. Allan Heydon and Marc Najork, “Mercator: A Scalable, Extensible Web Crawler”. Compaq Systems Research Center.
8. Boon Thau Loo and Sailesh Krishnamurthy and Owen Cooper. “Distributed Web Crawling over DHTs”. *UC Berkeley Technical Report UCB//CSD-4-1305*, Feb 2004.
9. Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, Hari Balakrishnan, “Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications.” *IEEE/ACM Transactions on Networking.*
10. Gnutella RFC.<http://rfc-gnutella.sourceforge.net>
11. Phex –P2P Gnutella File Sharing Program. <http://sourceforge.net/projects/phex>
12. Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. “Ubicrawler: A scalable fully distributed web crawler”. *Software: Practice & Experience*, 34(8):711–726, 2004.