PeerCrawl
A Decentralized Peer-to-Peer Architecture for Crawling the World Wide Web

VAIBHAV J. PADLIYA
MS CS Project (Fall 2005 – Spring 2006)

1 PROJECT GOAL

Most of the current web crawlers use a centralized client-server model, in which the crawl is done by one or more tightly coupled machines, but the distribution of the crawling jobs and the collection of crawled results are managed in a centralized system using a centralized URL repository. Centralized solutions are known to have problems like link congestion, being a single point of failure, and expensive administration.

Apoidea [1], project developed at Georgia Institute of Technology, is a completely distributed and decentralized Peer-to-Peer (P2P) architecture for crawling the World Wide Web, which is self-managing and uses geographical proximity of the web resources to the peers for a better and faster crawl. It uses Distributed Hash Table (DHT) based protocols to perform the critical URL-duplicate and content-duplicate tests. Peer lookups are done using Chord [2] based protocol which constitutes the network layer. Although, Apoidea manifests good crawling results, it has limited performance due to the protocol overheads incurred during information exchange across the peers and tight coupling between the Chord layer and the crawling layer.

This project focuses on improvising Apoidea for speed and efficiency. PeerCrawl is a distributed and decentralized P2P web crawler and uses Gnutella [11] protocol for formation of the network layer. The main advantages of PeerCrawl are as follows:

- **Scalability** PeerCrawl has been developed to be scalable. New peers can be added to the network on the fly to improve throughput.

- **Fully Decentralized** PeerCrawl is completely decentralized. This prevents any link congestion that might occur because of the communications with a central server. The peers perform their jobs independently and exchange information whenever they need to.

- **Dynamic** PeerCrawl is a self-managing system and requires no manual administration during entry and exit of peers.

- **Fault Tolerance** Since PeerCrawl takes into consideration entry and exit of peers, the system, by design, is fault tolerant. On Peer failure URLs are dynamically distributed among other peers. Besides, it has inherent redundancy, an important characteristic of distributed systems.

- **Efficient** Lesser overheads during information exchange make PeerCrawl faster and more efficient than Apoidea.
2 SYSTEM DESIGN

The distributed web crawler system involved a number of design issues as discussed below. The system consists of two major separable and loosely coupled components – overlay network and the crawler itself. The loose coupling ensures that the crawler can be ported to other Gnutella based overlay networks in future.

Overlay network
PeerCrawl uses Phex [12], an open source P2P file sharing client based on Gnutella protocol, for building an overlay network. Phex provides the necessary interfaces for network formation and maintenance. Phex has support for supernode architecture, a common design feature of P2P systems. However, this prototype assumes a flat architecture wherein all nodes have equivalent capabilities. The crawler is built on this layer and makes appropriate calls to Phex routines as needed.

Division of labor
For crawling purposes, the web is divided among the peers in the network; each peer is responsible for crawling a part of the web. A URL distribution function determines the domains to be crawled by a particular peer. This function is dynamic and the crawl range is recomputed, when peers enter and exit the system, for redistributing the load among remaining peers.

Duplicate checking
Even assuming each peer crawling a distinct part of the web, they will still encounter duplicate URLs and duplicate content. Ensuring that a URL is crawled only once is a major design issue since duplication testing prevents redundant crawling saving the crawler a considerable amount of time and resources. This prototype, however, considers only URL duplication testing on a single peer and uses bloom filters for the same.

Peer communication
Peers need to broadcast URLs not in their crawl range and maintain a count of nodes on horizon for dynamic adjustments to the crawl range. This communication is done though Gnutella query messages using Phex routines.

Data structures
Every peer needs to maintain data structures for crawling. These include:
- Crawl-Jobs: A list of URLs to be crawled.
- Seen-URLs: A list of URLs already crawled, used for duplicate URL testing.
- Send-URLs: A list of URLs to be broadcasted.

The biggest complexity is the selection and maintenance of these data structures, since the web crawler is memory intensive. Although, multiple threads operate on these data structures, they grow rapidly with time and can crash the system if left unbounded. Controlling these structures is necessary since the incoming rate for the lists far exceeds the outgoing rate.
**Multithreading**
For better performance and efficiency the crawl job is sub divided into smaller sub-jobs and each sub-job is performed by a separate thread. Further, multiple threads of each type are executed for better CPU utilization.

**Log generation**
The crawler generates logs for performance evaluation and future refinements. The logs include, among other data, http data, per-second and per-minute crawl statistics, peers connected in the network and URLs crawled.

### 3 SYSTEM ARCHITECTURE

The system primarily consists of two layers – network layer and the crawler itself. The network layer is responsible for network formation and management, communication between peers and routing. The crawler is responsible for determining the crawl range for the peer, checking the membership of the URL for the peer, crawling URL and extracting new URLs from downloaded page.

**Network layer**
An overlay network of peer crawlers is formed based on the Gnutella protocol. The open source P2P file sharing client Phex provides the necessary interfaces for network formation and management. This prototype has a flat network architecture wherein all peers have equivalent capabilities.

*Network formation*
This prototype supports both local host caching and web caching for network formation and network entry purposes.

The host cache is a list of peers in the network that each peer maintains locally. When a peer boots, it loads the list from the host cache and attempts to connect to those peers for entering the network. On exit the list is saved locally for re-entry at a later stage.

The web cache is a script placed on a web server that stores IP addresses of peers in the network and URLs of other web caches. Peers connect to a cache in their list randomly and receive IP addresses and URLs from the cache. Peers periodically send updates to the cache so that the caches remain fresh and eventually learn about each other.

The initial peer boots and listens for connections from other peers. The second peer connects to this peer using its host cache and the crawl network is formed. The peers send their IP addresses to the web cache. Hereafter, other peers can enter the network using either the host cache or the web cache or both.
Crawler layer
PeerCrawl consists of a number of peers all across the WWW. Each peer is responsible for a certain portion of the web. The first peer in the network starts crawling from the “seedURL”. Other peers joining the network catch broadcast URLs and start the crawl if URL is in range.
URL distribution, data structures and worker threads used by a single peer, the interfaces used by this layer for communicating with the underlying network layer and log generation are discussed in the following subsections.

Storage
The crawler uses the following data structures:

- **crawljobs** – list of URLs yet to be crawled by the peer. It is maintained as a hashtable with domains as the keys and the URLs belonging to that domain as the corresponding values.
- **pending** – list of URLs extracted from the downloaded pages and yet to be processed.
- **dispatch** – list of URLs to be broadcasted.
- **URLinfo** – list of URLs which have already been crawled. This is used to prevent the same URL being crawled again. It is maintained as a bloom filter.

These data structures are bounded to prevent uncontrollable growth.

- **crawljobs**: On reaching the max_domain limit the structure is cleaned to remove the domains with no URLs to be crawled. If this clean up does not create empty slots new domains are dropped but URLs for existing domains can still be added to the list. When the total URL count in the structure reaches the max_URL limit, all URLs are dropped until the min_URL limit is reached.

- **pending**: On reaching the max_size limit all URLs are dropped until the min_size limit is reached.

- **URLinfo**: On reaching the max_domain limit this structure is cleaned to remove the domains that have been removed from crawljobs structure. If the clean up does not create empty slots new domains are dropped.

URL Distribution
Each peer is responsible for crawling a range of URL domains. The range is determined by the distribution function. The IP address of peers and URL domains are hashed to the same 128-bit value space.

The domain range of a peer $p_i$ is computed as:

$$\text{upperbound}(p_i) = \min(h(p_i) + 2^{128 \cdot x} - 1, 2^{128} - 1)$$
$$\text{lowerbound}(p_i) = \max(0, h(p_i) - 2^{128 \cdot x})$$
$$\text{Range}(p_i) = \text{upperbound}(p_i) - \text{lowerbound}(p_i)$$

where,
- $h$ is the hash function
- $x$ (offset) depends on the number of peers in the system.

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. The receiving peers drop the URL if it’s not in range.
The distribution of URLs among peers is dynamic and based on number of peers on the horizon (horizon count) for a particular peer. As peers enter and exit the network, the peer’s horizon count changes and its range is recomputed to redistribute the load amongst the available peers. Every peer in the network maintains this horizon count by sending ping queries periodically to check which peers are still “alive”. Horizon count is adjusted based on the number pong replies received.

For a peer $i$ having horizon count of $(n-1)$, offset $x$ is computed as:

$$x = \log_2(n)$$

If $n$ is not a power of 2, $x$ can be taken as either $\text{floor}(x)$ (if URL duplication is preferred) or $\text{ceil}(x)$ (if URL duplication is not preferred).

Worker threads
As discussed in the system design, the core crawl job is divided into sub-jobs and each sub-job is performed by a separate thread.

- **Fetch** The fetch thread picks up a domain from the crawljobs list and crawls the pages belonging to that domain. Each domain is assigned to a single peer. This significantly speeds up the crawl since that peer uses the Connection-Alive feature of the HTTP protocol, in which a single open socket can be used to download a number of pages. The Robot Exclusion Protocol is used to prevent crawling pages from web sites that do not wish to be crawled. URLs extracted from the downloaded page are added to the pending list.

- **Batch** The batch thread picks up a URL from the pending queue and determines the domain membership for current peer. If the domain is not in current peer’s range, it is added to the broadcast list. Otherwise, URL is tested for duplication and added to the crawljobs list

- **Broadcast** This thread picks up URLs from the broadcast list, packages URL in the query message and broadcasts it.

Interfaces
The peers have to communicate with each other for crawling and network management. For this the crawl layer communicates with the network layer through interfaces provided by Phex. Peer communication takes place in following scenarios:

- **Broadcast**: If a peer extracts a URL belonging to a domain it is not responsible for, the URL needs to be broadcasted. This is done by invoking Phex query routine that packages the URL and sends it out.

- **Process query**: When a peer receives a query message, it is passed to the query processing routine in the crawler. URL is extracted from the payload field and tested for domain
membership. If the URL’s domain belongs to this peer, it is added to the crawljobs list. Otherwise, the URL is dropped.

- Horizon count: Phex maintains a count of peers on its horizon by sending ping queries periodically. This count is communicated back to the crawler for re-computing the peer’s crawl range.

4 RESULTS AND OBSERVATIONS

Experiments were conducted for evaluating the performance of PeerCrawl under different test conditions. The test bed consisted of identical Intel Pentium 4 1.8GHz machines having 512MB RAM.

The first experiment involved testing the performance of single peer crawler for varying number of threads. PeerCrawl was run on a single machine and URLs crawled per minute was recorded for 2, 5, 8 and 11 threads. As shown in figure 2, the crawl rate (per minute) increases with number of threads. The curve starts flattening a bit as the number increases.

![Figure 2](image)

The second experiment was conducted to test the performance of the crawler in distributed environment with varying number of peers in the network. PeerCrawl was run on the machines described above and global URLs crawled per minute was recorded with 1, 2, 4 and 8 peers in the network. As shown in figure 3, the total number of URLs crawled per minute increases with the number of peers in the network. Thus the throughput keeps increasing as more and more peers join the network.
The third experiment shows the contrast between the first prototype of PeerCrawl with unbounded buffers and second (current) prototype with bounded buffers. As shown in figure 4, first prototype had higher crawl rate and zero drop rate, while second prototype has lower smaller crawl rate and high drop rate. While first prototype crashed in about 30 minutes, second prototype crawled for more than 5 hours without crashing. This shows that current prototype is more stable, though it has a lower crawl rate compared to first prototype. PeerCrawl’s crawl rate is still significantly higher than that of Apoidea.
5 LOGS AND GRAPHICAL DISPLAY

The crawler generates logs for performance evaluation and future refinements. The logs include, among other data, http data, per-second and per-minute crawl statistics, peers connected in the network and URLs crawled. These logs are saved in log files saved directly to the disk. The files are verbose and an independent graphical analyzer can be built to analyze these files.

This prototype encompasses two forms of graphical displays.

- An integrated, non-interactive, text based display that outputs statistics every second and every minute.
- An interactive, standalone graphical display which uses crawler log files as input. It can run in online (while crawler is running) and offline (crawler is not running) modes.

Figures below illustrate a few screenshots of both types of displays.

![Figure 5](image)

Figure 5 above depicts the text based display showing the URLs already crawled.
As shown in Figure 6, the user can check whether a particular URL has been crawled. The user can also check the number of URLs crawled in a particular domain. The graph depicts the number of URLs crawled every second.

Figure 7

Figure 7 depicts the number of URLs added to the crawl job queue. Figure 8 shows the content crawled so far. As rightly projected, most of the data on the internet is text/html.
6 RELATED WORK

Most of the crawlers have centralized architecture. Google [6] and Mercator [7] have single machine architecture. They deploy a central coordinator that dispatches URLs. [5, 6] presents a design of high performance parallel crawlers based on hash based schemes. These machines suffer from drawbacks of centralized architecture discussed above. [1, 10] 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 address 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.

7 CONCLUSION AND FUTURE WORK

The project describes the design and development of a fully decentralized and distributed web crawler based on Gnutella protocol. Two prototypes have already been developed and tested for performance. From the results, it is clear that the second prototype is more stable than the first prototype and faster and more efficient as compared to Apoidea. Current PeerCrawl version crawls about 2000 URLs per minute with a network of 8 peers. Considering WWW has 1 billion web pages current PeerCrawl can crawl the web in approximately 11 months, 12 days, 5 hours and 20 minutes.
PeerCrawl though stable and efficient is in its inception stage. The prototype can be improvised by incorporating many more functionalities and features of a distributed system. These include:

- **Supernode architecture** The current prototype has flat network architecture. All the peers are at the same level. A major shortcoming of this approach is uncontrolled flooding among the peers. This can be overcome by using “supernode architecture” wherein leaf peers communicate only with the “supernode” they are connected to while supernodes flood data. This architecture has scope for controlled crawling and better load balancing among the peers.

- **Geographical proximity** PeerCrawl does not exploit geographical proximity of resources for crawling. For example, a peer in India would deliver better crawl rates while crawling a “.in” domain as against a peer in US. The URL distribution function can be improvised to address this issue.

- **Buffer bounds and URL Ranking** Currently, the buffers in PeerCrawl are hard bounded and all URLs are dropped after the upper limits are reached. The technique can be refined to allow “intelligent discrimination” of URLs for dropping purposes. One approach would be to rank the URLs and drop the ones with lower ranks. This would ensure crawling better URLs and better overall performance of the crawler. Further the buffer bounds were set on a trial and error basis. Tuning the bounds based on the system and empirical data would further enhance the results.

- **User level thread control** Major tasks in crawler are split among multiple threads. However, in this prototype the threads are not fully user controlled and thus relying on the underlying JVM for thread management. User control over threads would help optimize memory requirements.

- **Replacing network layer** The current prototype uses Phex for providing the network layer. In future, the crawler can be ported to other P2P clients for testing the performance.
References


