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

Aameek Singh, Mudhakar Srivatsa, Ling Liu, and Todd Miller

College of Computing,
Georgia Institute of Technology,
Atlanta, GA - 30332
{aameek,mudhakar,lingliu,tomiller}@cc.gatech.edu

Abstract. This paper describes a decentralized peer-to-peer model for building a Web crawler. Most of the current systems 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. It requires both horizontal and vertical scalability solutions to manage Network File Systems (NFS) and load balancing DNS and HTTP requests.

In this paper, we present an architecture of a completely distributed and decentralized Peer-to-Peer (P2P) crawler called Apoidea, which is self-managing and uses geographical proximity of the web resources to the peers for a better and faster crawl. We use Distributed Hash Table (DHT) based protocols to perform the critical URL-duplicate and content-duplicate tests.

1 Introduction

Search engine technology has played a very important role in the growth of the WWW. Ability to reach desired content amidst huge amounts of data has made businesses more efficient and productive. An important component of this technology is the process of *crawling*. It refers to the process of traversing the WWW by following hyperlinks and storing downloaded pages. Most of the currently available web crawling systems [5,10] have envisioned the system as being run by a single organization. The architecture of such a system is based on a central server model, which determines which URLs to crawl next and which web pages to store and which to dump (in case, a similar page is already available in the database by maybe, crawling a mirror site).

Such a system requires organizations to employ extremely resourceful machines and various experienced administrators to manage the process. For example, Mercator [10] used a machine with 2 GB of RAM and 118 GB of local disk. Google [5] also utilizes a special hardware configuration and centralized management of tens and thousands of machines to crawl the web. Along with

J. Callan et al. (Eds.): SIGIR 2003 Ws Distributed IR, LNCS 2924, pp. 126–142, 2003.

higher costs, an important problem arises because of the congestion of the link joining the crawling processes with the central server, where all the data is kept.

With the emergence of successful applications like Gnutella [9], Kazaa [11], and Freenet [7], peer-to-peer technology has received significant visibility over the past few years. Peer-to-peer (P2P) systems are massively distributed computing systems in which peers (nodes) communicate directly with one another to distribute tasks or exchange information or accomplish tasks. P2P computing has a number of attractive properties. First, it requires no additional hardware, and the computing resources grow with clients. Second, it is self-organizing, incurring no additional administrative costs due to scaling. Third but not the least, it is fault-tolerant by design. P2P systems differ from each other in terms of how they exchange data or distribute tasks, and how they locate information (lookup service) across the overlay network.

In this paper we present a peer to peer approach to crawl the web. Apoidea[1] is a fully decentralized P2P system with no central authority, much like the family of bees it derives its name from. All the bees (peers) work independently without any commanding authority to decide further crawling and the data is kept distributed across the network. The task of crawling is divided amongst the peers and we use DHT based distributed lookup and information-exchange protocols to exchange vital information between the peers. Also, we use bloom filters [3] to store the list of URLs already crawled by a peer. This makes the memory requirements at each peer very reasonable and easily available at normal PCs of today.

The option of distributing the task of crawling across the internet also provides us of an opportunity to exploit geographical proximity of crawlers to the domains they are crawling. Our initial results indicate that using such a feature can significantly speed up the crawling process. Also having a decentralized system makes it possible to crawl the web with minimal or no supervision and using well studied replication mechanisms, such a design is inherently more fault tolerant.

Please note that in this paper, we focus only on the crawling of the WWW. We do not present any mechanisms for indexing and providing a complete search engine. These mechanisms have been well studied and can be used on top of our design architecture.

The main advantages of Apoidea are as follows:

- **Efficient**: Apoidea utilizes peers geographically closer to the web resources to crawl. This can lead to significant speed improvements and greater efficiency. It also serves as an automatic load balancing mechanism for DNS servers and local proxy servers.
- **Fully Decentralized**: Apoidea 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.

[1] Apoidea is a family of bees which does not have any queen bee

- **Low Cost**: Apoidea is a self-managing system. It means there is no need of manual administration. The system automatically handles the dynamics of P2P systems - the entry and exit of peers. Also the resource requirements at every peer are very reasonable.
- **Greater Reach**: With the increase in the number of peers in the system, we can potentially reach a much greater part of the WWW as opposed to what conventional systems can do.
- **Scalability**: Apoidea has been constructed with an aim of scalability in mind and we envision the system to work efficiently with huge WWW growth rates.
- **Fault Tolerance**: Since Apoidea takes into consideration entry and exit of peers, the system, by design, is more fault tolerant than the ones available today, which have a single point of failure.

The rest of the paper is organized as follows:
Section-2 describes the basics of crawler design and a look at DHT based distributed lookup protocols and bloom filters. *Section-3* describes the complete architecture and working details of Apoidea. In *Section-4*, we present our initial results and observations of the performance of the system. In *Section-5*, we discuss how Apoidea could be used to build a World Wide Web search engine. In *Section-6*, we discuss the work related to this paper. Finally, *Section-7*, contains our conclusions and future directions of research.

2 System Design: The Basics

In this section, we briefly describe general design requirements for a web crawler. Then we will talk about the work done in the area of DHT based P2P systems and bloom filters and how we intend to use them in our architecture. Later, in Section-3 we concretely explain how the whole system is put in place and its workflow details.

2.1 Design Considerations

In this section, we describe the basic design components of any crawler and specifically the important design principles that have to be taken into consideration for distributed and decentralized crawlers.

A typical web crawler consists of three primary components:

- *Downloader*: This component reads a list of URLs and makes HTTP requests to get those web pages.
- *URL Extractor*: This component is responsible for extracting URLs from a downloaded web page. It then puts the URL in the to-be-crawled list.
- *Duplicate Tester*: This component is responsible for checking for URL and content duplicates.

For the crawl, it also maintains the following data structures:

- *Crawl-Jobs*: It is the list of URLs to be crawled.
- *Seen-URLs*: It is a list of URLs that have already been crawled.

– *Seen-Content*: It is a list of fingerprints (hash signature/checksum) of pages that have been crawled.

The downloader picks up URLs from the Crawl-Jobs data structure and downloads the page. It is checked for a content duplication. If it turns out to be a duplicate, it is thrown away. If it is a new page, its fingerprint is added to the Seen-Content data structure. The URL Extractor component then processes this page to extract more URLs and normalizes them. The URLs are then checked for possible duplicates and if found new, are added to the Crawl-Jobs list. Along with it, they are also added to the Seen-URLs list. Both the duplicate tests have been proved to be important. Encountering a duplicate URL is extremely common, due to popularity of content and a website having common headers and footers for all its web pages. Duplicate pages occur due to mirror sites and also when there are symbolic links at a web server, for example, both home.html and index.html point to the same page on many web servers. In Mercator's [10] experiments, around 8.5% of the pages were duplicates.

For a distributed crawler, decisions have to made for selecting which crawler gets to crawl a particular URL. This can be as simple as just picking up URLs from Crawl-Jobs sequentially and giving it to the crawlers in the round robin fashion or can be based on a complicated policy. For example, Google [5] uses various machines in a round robin manner and the Internet Archive [6] crawler distributes them based on domains being crawled.

However, designing a decentralized crawler has many new challenges.

1. **Division of Labor**: This issue is much more important in a decentralized crawler than its centralized counterpart. We would like the distributed crawlers to crawl distinct portions of the web at all times. Also there are potential optimizations based on geographical distribution of the crawlers across the globe. In Section-3, we describe how we exploit the global presence of the peers in the Apoidea system.

2. **Duplicate Checking**: Even assuming each peer crawling a distinct part of the web, they will still encounter duplicate URLs and duplicate content. Since there is no single repository of Seen-URLs and Seen-Content data structures, the peers have to communicate between each other and decide on the newness of a URL or a page. And it is extremely important to keep these communication costs to a minimum since we can potentially have thousands of peers across the globe crawling the web at a very fast speed. We use DHT based protocols, which achieves these lookups in $O(log\ n)$ time, where n is the total number of peers.

It is important to notice that the above two components are not entirely disconnected from each other. Having a good division of labor scheme can potentially save us on the duplicate decision making part.

2.2 Crawling the Web Using a Peer to Peer Network

In the recent past, most of the research on P2P systems was targeted at improving the performance of search. This led to the emergence of a class of P2P systems that include Chord [17], CAN [13], Pastry [14] and Tapestry [2]. These techniques are fundamentally based on distributed hash tables, but slightly differ in algorithmic and implementation details. All of them store the mapping between a particular *key* and its *value* is a distributed manner across the network, rather than storing them at a single location like a conventional hash table. This is achieved by maintaining a small routing table at each node. Further more, given a *key*, these techniques guarantee the location of its *value* in a bounded number of hops within the network. To achieve this, each peer is given an identifier and is made responsible for a certain set of keys. This assignment is typically done by normalizing the key and the peer identifier to a common space (like hashing them using the same hash function) and having policies like numerical closeness or contiguous regions between two node identifiers, to identify the regions which each peer will be responsible for.

For example, in the context of a file sharing application, the key can be a file name and the identifier can be the IP address of the peer. All the available peers' IP addresses are hashed using a hash function and each of them store a small routing table (for example, for Chord the routing table has only m entries for an m bit hash function) to locate other peers. Now, to locate a particular file, its filename is hashed using the same hash function and depending upon the policy, the peer responsible for that file is obtained. This operation of locating the appropriate peer is called a *lookup*.

A typical DHT-based P2P system will provide the following guarantees:

- A lookup operation for any key k is guaranteed to succeed if and only if the data corresponding to key k is present in the system.
- A lookup operation is guaranteed to terminate within a small and finite number of hops.
- The key identifier space is uniformly (statistically) divided among all currently active peers.
- The system is capable of handling dynamic peer joins and leaves.

Apoidea uses a protocol similar to that of Chord [17], because of its simplicity in implementation. In Apoidea, we have two kinds of keys – URLs and Page Content. The term – *a peer is responsible for a particular URL*, means that the URL has to be crawled by that particular peer and for the page content, it means that the peer has information whether that page content has already been downloaded (probably by crawling a mirror site) or not.

Now, a URL lookup can be done as follows. First, we hash the **domain name** of the URL. The protocol performs a lookup on this value and returns the IP address of the peer responsible for this URL. Note that this choice will result in a single domain being crawled by only one peer in the system. It was designed in this manner, since now a peer can use the *Connection-Alive* parameter of the HTTP protocol and use a single TCP/IP socket while downloading multiple

pages from a particular domain. This significantly fastens the crawl, because we save on the costs of establishing a TCP/IP connection with the domain, for every URL crawled, which would have been the case if we had just hashed the complete URL and done a lookup on it.

The page-content is used as a key when we intend to check for duplicate downloaded pages. For this, the page-content is hashed and a lookup is initiated to get the peer responsible for this page-content. Note that, the page could (and in most case, would) have been downloaded by another peer (since that is based on the lookup of the URL's domain name). The peer responsible for this page content only maintains the information and does not store the entire page.

Also, because of the P2P nature, the protocol must handle dynamic peer join and leave operations. Whenever a new peer joins the system, it is assigned responsibility for a certain portion of the key space. In the context of a web crawler, the new peer is assigned those domains that hash onto it. The peers that were currently responsible for those domains are required to send information about the list of URLs already crawled in those domains to the newly entering node. A very similar technique is used to redistribute the page signature information.

When a `Peer` P wants to leave the network, it sends all URL and page content information currently assigned to it to other nodes. However, if the `Peer` P were to fail, then it would not be able to send any information to other peers; that is the information about the keys held by `Peer` P would be temporarily lost. This problem is circumvented by replicating information. Every domain D is mapped to multiple keys $(k_1, k_2, \cdots, k_R)$ which are in turn handled by distinct peers. However only the peer responsible for key k_1 (*primary replica*) is responsible for crawling the domain D. The peers responsible for the secondary replicas (keys $k_2, k_3, \cdots, k_R$) would simply store information as to which URLs in domain D have been already crawled or not. However, if the peer responsible for the primary replica crashes, the information held by it could be recovered using one of its secondary replicas. For the sake of simplicity, we will consider only having one key for each domain in the rest of the paper.

2.3 Managing Duplicate URLs Using Bloom Filters

Bloom filters provide an elegant technique for representing a set of key elements $K = \{k_1, k_2, \cdots, k_n\}$ to support membership queries. The main idea is to allocate a vector of m bits, initially set to 0 and choose l independent hash functions $h_1, h_2, \cdots, h_l$ with range $\{0, 1, \cdots, m-1\}$. For any key $k \in K$, the bit positions $h_1(k), h_2(k), \cdots, h_l(k)$ is set to one. Note that a particular bit location may be set to one by more than one key. Now, a membership check is performed as follows. Given a key k, we check for bits at positions $h_1(k), h_2(k), \cdots, h_l(k)$ in vector m. If any of them is zero, then certainly $k \notin K$. Otherwise, we assume that the given key $k \in K$ although our conclusion could be wrong with a certain probability. This is called a *false positive* .

The salient feature of the Bloom Filters is in the fact that one can trade off space (m) with the probability of a false positive. It has been shown in [3] that using $l = ln2 * m/n$ independent hash functions, one can reduce the probability

of false positive to $(0.6185)^{\frac{m}{n}}$. We can achieve a probability of false positive as low as 0.001 for $l = 5$ independent hash functions and $m/n = 10$.

In the context of a web crawler, we can use a bloom filter to maintain the information about a URL having been crawled or not. All the URLs that have already been crawled form the set K. Now, to test a URL for its newness, we hash it to a value k and then check for membership of k in K. This will indicate whether that URL has already been crawled or not. Assuming that the WWW contains about 4 billion pages and setting $m/n = 10$ the size of our bloom filter would be $4 * 2^{30} * 10$ bits or 5GB. Note that any practical implementation of the bloom filters would like to maintain the entire filter in its main memory. Hence, it is infeasible to hold the entire bloom filter required for the scale of the WWW in a single machine. In our design we divide the URLs in WWW amongst many cooperating peers. Assuming that we have only a 1000 cooperating peers in different geographical locations then each of these peers would have to handle about 4 million web pages, which would in turn require a bloom filter of size 5MB. Given today's technology, each peer can easily afford to maintain this 5MB bloom filter in its main memory.

3 System Architecture

Apoidea consists of a number of peers all across the WWW. As explained in the previous section, each peer is responsible for a certain portion of the web. In this section we first describe how we perform the division of labor and how the duplicate checking is done in Apoidea. Then we briefly describe the peer-to-peer (P2P) architecture of Apoidea, especially the membership formation and the network formation of peers, and the single peer architecture in Apoidea, including how each peer participates in the Apoidea web crawling and performs the assigned crawling jobs.

– **Division of Labor:** As we have already discussed in Section-2, we use a DHT-based system for distributing the WWW space among all peers in the network. A `Peer` P is responsible for all URLs whose domain name hashes onto it. In Apoidea, we do it using the contiguous region policy, which will be clearer from the example shown in Figure-1. Assume that there are three peers in the system - `Peer` A, B and `C`. Their IPs are hashed onto an m bit space and hence, will be located at three points in the modulo m ring. Then, various domain names are also hashed onto this space and they will also occupy such slots. The `Peer` A is made responsible for the space between `Peer` C and itself. `Peer` B is responsible for space between `Peer` A and itself and `Peer` C with the rest. From the figure, `Peer` A is responsible for the domain *www.cc.gatech.edu*. Therefore `Peer` A will be the only peer crawling all the URLs in that domain. If any other peer gets URLs belonging to that domain, it batches them and periodically send them to `Peer` A.
  However, such a random assignment of URLs to peers does not exploit the geographical proximity information. In order to ensure that a domain

is crawled by a peer *closer* to the domain, we relax the tight mapping of a domain to a peer as follows. Every Peer P maintains a list of *leaf peers* that are close to Peer P in the identifier space. In the context of the Chord protocol, a Peer P maintains pointers to $l/2$ successor peers and $l/2$ predecessor peers on its identifier ring. Now, when a Peer P is assigned to domain D, it picks the leaf peer closest to domain D and forwards the batch of URLs to it for crawling.

– **Duplicate Checking:** Let us assume that Peer P is responsible for domain D and that Peer Q a leaf peer of Peer P is actually crawling domain D. When any peer encounters a URL in domain D, it sends the URL to Peer P since it is responsible for domain D. Now, Peer P batches a collection of URLs in domain D and forwards them to Peer Q. Recall that Peer Q maintains a bloom filter which indicates as to whether a URL is already crawled or not. Hence, Peer Q checks if a given URL is already crawled or not. If the URL were indeed already crawled then it is simply dropped, else it is added to the Crawl-Jobs list.
In order to check page duplication, we hash the page contents onto the identifier space and hence distribute them amongst the various peers in the network. Figure-2 shows an example of it. The page content of *www.cc.gatech.edu/research* is hashed onto the ring and Peer C is responsible for it. Note that the page would have been crawled by Peer A because the domain *www.cc.gatech.edu* hashed onto it. When Peer A encounters that page, it needs to check whether that page is a duplicate or not. So it looks up for the hash of page content and finds out Peer C is responsible for it. Now Peer A can batch all such requests and periodically query Peer C about the newness. Peer C can then batch the replies back. It also modifies its local Seen-Content list to note the new pages that have been downloaded by Peer A. Note that the delay in getting the information about the duplicity of a page is not performance-effecting since always, there is a significant delay in downloading a particular page and processing that page. This is because the downloads typically happen at a much faster rate than the processing and as a result the processing has a significant lag from the download.

3.1 Apoidea Peer-to-Peer Network Formation

Peers in the Apoidea system are user machines on the Internet that execute web crawling applications. Peers act both as clients and servers in terms of their roles in performing crawling jobs. An URL crawling job can be posted from any peer in the system. There is no scheduling node in the system and neither does any peer have a global knowledge about the topology of the system. There are three main mechanisms that make up the Apoidea peer to peer (P2P) system.

1. **Peer Membership:** The first mechanism is the overlay network membership. Peer membership allows peers to communicate directly with one

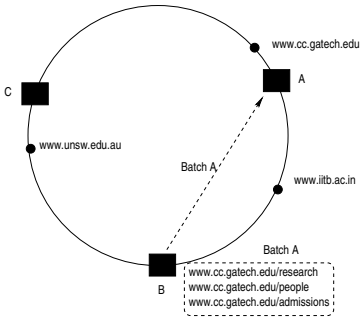

Fig. 1. Division of Labor

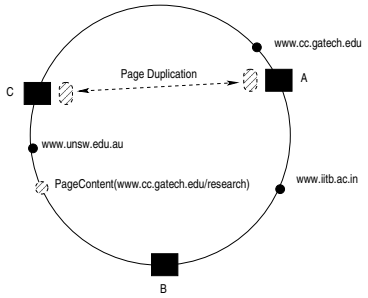

Fig. 2. Content-duplicate checking

another to distribute tasks or exchange information. A new node can join the Apoidea system by contacting an existing peer (an entry node) in the Apoidea network. There are several bootstrapping methods to determine an entry node. We may assume that the Apoidea service has an associated DNS domain name. It takes care of resolving the mapping of Apoidea's domain name to the IP address of one or more Apoidea bootstrapping nodes. A bootstrapping node maintains a short list of Apoidea nodes that are currently alive in the system. To join Apoidea P2P Web crawling, a new node looks up the Apoidea domain name in DNS to obtain a bootstrapping node's IP address. The bootstrapping node randomly chooses several entry nodes from the short list of nodes and supplies their IP addresses. Upon contacting an entry node of Apoidea, the new node is integrated into the system through the Apoidea protocol's initialization procedures. We will revisit the process of entry and exit of nodes in Section-3.3.

2. **Protocol:** The second mechanism is the Apoidea protocol, including the partitioning of the web crawling jobs and the lookup algorithm. In Apoidea every peer participates in the process of crawling the Web, and any peer can post a new URL to be crawled. When a new URL is encountered by a peer, this peer first determines which peer will be responsible for crawling this URL. This is achieved through a lookup operation provided by the Apoidea protocol.

   The Apoidea protocol is in many ways similar to Chord [17]. It specifies three important types of peer coordination: (1) how to find peers that are best to crawl the given URL, (2) how new nodes join the system, and (3) how Apoidea manages failures or departures of existing nodes. A unique feature of the protocol is its ability to provide a fast distributed computation of a hash function, mapping URL crawl jobs to nodes responsible for them.
3. **Crawling:** The third mechanism is the processing of URL crawling requests. Each URL crawling job is assigned to a peer with an identifier matching the domain of the URL. Based on an identifier matching criteria, URLs are crawled at their assigned peers and cleanly migrated to other peers in the presence of failure or peer entrance and departure.

From a user's point of view, each peer in the P2P network is equipped with the Apoidea middleware, a two-layer software system. The lower layer is the Apoidea P2P protocol layer responsible for peer-to-peer communication. The upper layer is the web crawling subsystem responsible for crawling assigned URLs, resolving last-seen URLs to avoid duplicate crawling, and processing crawled pages. Any application-specific crawling requirements can be incorporated at this layer.

In the subsequent sections we discuss the Apoidea middleware that runs at every Apoidea peer to describe the design of the P2P crawling subsystem in more detail. We also report some initial experimental results obtained by evaluation of the first prototype of Apoidea P2P Web crawlers.

3.2 Single Peer Architecture

The architecture of every peer is composed of three main units:

1. **Storage**: This unit contains all the data structures that each peer has to maintain.
2. **Workers**: This unit consists of modules fetching pages from the WWW and processing them to extract more URLs.
3. **Interface**: This unit forms the interface of each peer, handling communications within the Apoidea network.

The complete architecture is shown in Figure-3.

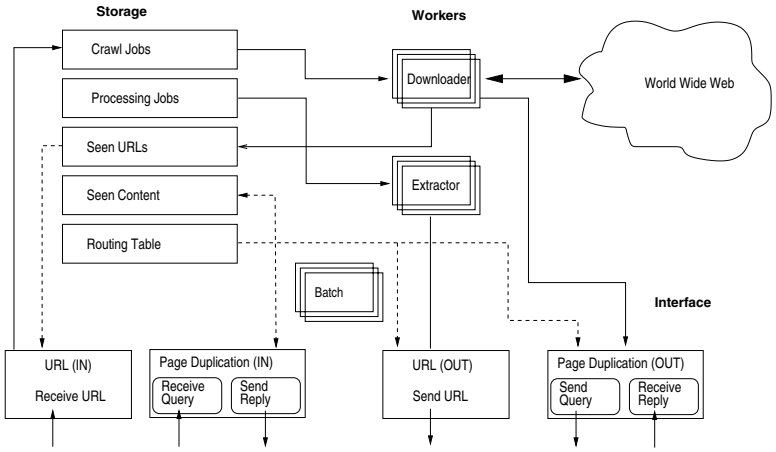

Fig. 3. System Architecture for a Single Apoidea Peer

Storage: This unit contains all the data structures maintained within each peer. These are:

- *Crawl-Jobs*: This is a list of URLs which are yet to be crawled by the peer. Since we batch URLs to a peer domain-wise, we maintain it as a hashtable with domains as the keys and the URLs belonging to that domain as the corresponding values.
- *Processing-Jobs*: This is a list of downloaded pages which are yet to be processed. The list is just a Vector of pointers to the pages. After a certain number of pages, the rest are stored on disk.
- *Seen-URLs*: This is a list of URLs that have already been crawled. This is used to prevent a same URL being crawled again. We maintain it as a bloom filter.
- *Seen-Content*: This is a list of page signatures (hash), which have already been seen by some peer in the network. An important point to be noted about this data structure of the peer is that it contains the page signatures for which this peer is *responsible for* and not the pages it downloads. Again, we use a bloom filter for this data structure.
- *Routing Table*: This is used to route the lookup queries to the right peer and it contains information about a small number of nodes in the system [17].

The biggest complexity in this component is the choice of data structures, since the web crawler is very extensive on resources, especially memory. The Crawl-Jobs and Processing-Jobs lists are usually easily maintainable since there are a number of downloaders and extractors working on them. Also the Routing-Table is a very small data structure. The main requirement is to efficiently manage the Seen-URLs and Seen-Content data structures. These data structures increase in size with the passage of time.

In order to maintain these data structures, we use bloom filters. Assume that the WWW has about 4 billion web pages. From our experiments we observed that each domain has about 4K URLs. Hence, the number of domains in the WWW would be about 1 million. Now, assuming that 1K peers participate in the crawling operation, each peer would crawl about 1K domains. For each of the domains we maintain a bloom filter that indicates whether a particular URL in that domain is already crawled or not. Noting the fact that these bloom filters need to hold about 4K URLs, we chose the size of the bloom filter to be 8KB so that the probability of false positive remains below one in a thousand (Using the formula in Section-2 and m = 8KB = 64K bits = 16 * 4K URLs). Every peer maintains a bloom filter of size 8MB (8KB per domain * 1K domains per peer). The peers also maintain the association between the domain name and its bloom filter in a hash table. Observing the fact that the average length of an URL is 80B and the hash table needs to hold 1K entries (1K domains) the size of this hash table would be about 80KB. Maintaining per domain bloom filters is essential for handling peer joins and leaves. Recall that every peer is assigned some domains. Hence, a newly joined node can construct its bloom filter for the domains assigned to it by simply copying the relevant bloom filter from the peer that was previously responsible for that domain.

Note that by using a bloom filter we have specifically avoided storing all the seen URLs in the memory. In fact, Mercator [10] uses this approach of

storing all the URLs in the domain it is currently crawling in its main memory. However, Mercator reports a hit rate of only 75%, which warrants several costly disk I/O operations. Such costly disk operations might be suitable when the peer is entirely dedicated to crawling. However, in a P2P scenario one would like to minimize those operations that can significantly affect the normal functioning of the peer. Also using a peer to crawl just a single domain at one time is inefficient since having a problem with that domain (say, shut down for maintenance) will significantly reduce the throughput of the system.

Workers: This unit mainly consists two sets of threads - *Downloader* and *Extractor*.

1. **Downloader:** The Downloader threads pick up a domain from the Crawl-Jobs list and queries l neighbors maintained in its neighbor list to find out the peer which can crawl that domain fastest. This can be done in very short message exchanges with each peer measuring the round trip time to that domain. If this peer is not the fastest peer, it can just pass on the URLs to the fastest neighbor. Usually this happens because of the geographical proximity of the neighbor to the domain being crawled. As our initial experiments indicate, this can be a very important factor in the total speed of crawling the whole WWW.
   The Robot Exclusion Protocol is taken into consideration before crawling any particular website, so that any site which does not wish to be crawled is not crawled. As mentioned before, 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.
   After a page is downloaded, it is stored in a temporary data structure and checked for page duplication. If this is a new page, it is inserted into the Processing-Jobs list, else dropped.

2. **Extractor:** The Extractor threads pick up pages from Processing-Jobs list and extracts URLs from that particular page. The URLs are extracted using the regular expression library of Java. After the extraction, the URLs are normalized (changing relative URLs to exact URLs, removing extraneous extensions like "javascript:", which cannot be handled by the crawler etc). These URLs are then kept in a temporary data structure from where they are periodically picked and batched to be submitted to the appropriate peers, that is, the peers who are responsible for them. This temporary data structure is a hash table with URLs belonging to the same domain in the same bucket.

Interface: While crawling, the peers have to communicate with each other in the following scenarios:

1. *Submission of URLs*: Peer A may extract URLs (from a page it crawled) which Peer B is responsible for, that is, the domain of those URLs is hashed

to be belonging to Peer B. Peer A batches these URLs and periodically send the URLs to Peer B. Then Peer-B simply ignores the URLs that is has already crawled else it adds them to its Crawl-Jobs list.
2. *Checking for Page Duplication*: After Peer A downloads a page, it has to check if a similar page is already available in the network or not. It does a lookup on the page hash and contacts the peer responsible for that hash value, say Peer B. In case Peer-B has the same page, Peer A will drop that page, else it stores it. Also Peer B notes that such a page is available in the network now.
3. *Protocol Requirements*: Such communications are required to maintain and stabilize the network. It includes periodic message exchange between neighboring peers to identify if a new peer has joined the network or some peer has exited the network.

We use low-cost UDP messages for the exchange of this information and a set of batching threads that periodically pick up batches of information for various peers and send the data.

3.3 Entry and Exit of Peers

In this section, we will illustrate how Apoidea handles dynamics associated with a P2P system - the entry and exit of peers.

Entry: Like any P2P based system, there will be a few bootstrap servers; at all times, there will be at least one which will be up and active. These servers act as entry points into the network. Whenever a peer wants to join the Apoidea network, it will contact a bootstrap server, which will point it to a node active in the network. Depending upon the IP address of the entering peer, the node will help it get the routing table required to participate in the network. Also the network stabilizes recognizing the entry of a new node. This procedure is inexpensive and can be easily accomplished using known methods provided by Chord [17].

During this stabilization process, the node which holds a list of Seen-URLs that the new peer is responsible for now, will send that data to the new peer. This data is just a short bloom filter for the domains that hash onto the new peer. Similar transfer will take place for the page content. After a peer has joined the network, it will periodically begin to get URLs of the domains it is responsible for and it will start the crawling process.

Exit: When a Peer P needs to leave the system, its Seen-URLs and Seen-Content have to be transferred to the peer which will be responsible for it now. The new responsible peer will always be a neighbor of Peer P on the ring space. This neighbor information is always stored at the peer as part of the routing table. Therefore, before its exit, it can just send the Seen-URLs and Seen-Content information to that neighbor.

In case of a peer failure, the network stabilization process corrects the routing tables of the peers in the network. However, we would have lost the information regarding Seen-URLs and Seen-Content. To prevent this, as mentioned in Section-2, we can maintain multiple replicas of this information. The meagre requirements of memory due to the use of bloom filters makes it possible for peers to act as primary as well as secondary replicas for various keys.

4 Results and Observations

We conducted experiments on geographical proximity and the scalability of Apoidea. Using a primitive Java implementation of the crawler, we crawled four similar domains (in terms of their networking infrastructure) from Georgia Institute of Technology: *.edu* domains starting form *www.cmu.edu* in USA, *.ac.jp* domains starting from *www.u-tokyo.ac.jp* in Japan, *.edu.au* domains starting from *www.unsw.edu.au* in Australia and *.ac.in* domains starting from *www.iitb.ac.in* in India. The results from our experiment are shown in Figure-4. Clearly, the figure shows that *.edu* domains being geographically the closest can be crawled at about twice the speed of the geographically farther domains like *.edu.au*. Given the fact that the networking infrastructure of these domains are comparable, a domain like *www.unsw.edu.au* can be crawled at about twice the speed from a geographically closer location. This, in turn, can have a huge impact on the total speed of the web crawl.

We performed two experiments on scalability. In our first experiment, we measured the resource requirements (CPU + Memory) on a single peer. We ran Apoidea on a Intel Pentium 550MHz machine under two scenarios: high load (running a ns2 simulator) and low load. Note that we intentionally chose a heavily loaded machine so as to restrict the resource utilization of Apoidea. Figure-5 shows the performance of Apoidea in terms of the number of URLs crawled per second with varying number of Java threads. Observe that when the machine was otherwise heavily loaded, Apoidea's performance does not increase with the number of threads for more than two threads (other than the JVM's overhead in handling more threads), since at any time most of the threads stay in the sleep state. Also note that under high load, Apoidea's CPU utilization was restricted to 3% and its memory consumption was limited to only 30MB. However, under lightly loaded conditions, the maximum rate is achieved when a peer employs six threads. Note that under lightly loaded conditions, Apoidea's CPU utilization was 30% and its memory consumption was 30MB.

In our second experiment on Apoidea's scalability, we increased the number of peers participating in the system. We experimented on a maximum of 16 machines on the same local area network in Georgia Institute of Technology. Note that the peers were either heavily loaded or lightly loaded as in described in our previous experiment. Also, each peer used two threads under heavy load and six threads under light load so as to obtain the best performance (from Figure-5). Figure-6 shows the performance of Apoidea with varying number of peers. Note that when only one peer was running Apoidea there is no cost of interaction

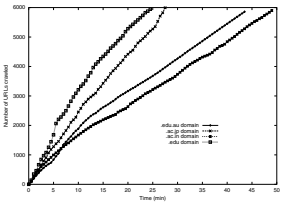

Fig. 4. Impact of Geographical Proximity

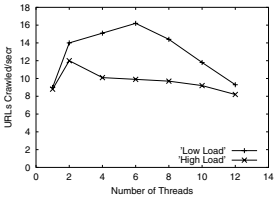

Fig. 5. Apoidea's Performance on one peer

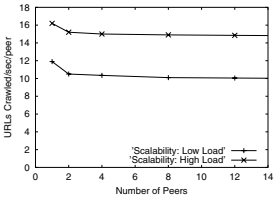

Fig. 6. Scalability of Apoidea with the Number of Peers

between the peers. Hence, adding the first peer lowers Apoidea's performance. However, note that with subsequent addition of peers, the performance almost flattens out even for small peer population of the order of 12 or 16 irrespective of the load on the peers.

In the near future, we propose to study the performance of Apoidea in the presence of a large peer population, possibly across multiple geographical locations. This will require volunteers from all over the world to run Apoidea. Second, we would like to measure the crawl refreshing speed; that is, having crawled the entire WWW once, we want to measure the speed at which the WWW can be crawled only for updates during the second and subsequent crawls. Third, we want to perform experiments on the reliability of the system and see the impact dynamic peer failures and replication can have on the overall network.

5 Application: A World Wide Web Search Engine

Several papers have proposed architectures to build scalable distributed indexes for keyword search over P2P networks [12,8]. Their designs are layered over distributed hash table based P2P systems like Chord [17]. [12], inspired by the ranking system used by Google, exploits statistics about popular documents and hosts in building distributed inverted indexes and answers a query by computing a join over the indexes matching the query keywords. On the other hand, [8] builds a distributed inverted index keyed on all combinations of multiple keywords. Thus [8] avoids the overheads of computing joins thereby saving on the network bandwidth at the cost of storage space (Note that the total storage space required grows exponentially with the set size of keyword combination used to build the inverted index).

Apoidea in conjunction with these distributed indexing architectures can be used to implement a search engine for the World Wide Web. We envision a distributed search engine wherein both Apoidea and distributed indexing architectures like [12,8] co-exist on any node in the P2P network. Apoidea crawls the World Wide Web and extracts relevant keywords from web pages. The indexing architecture builds a distributed inverted index and evaluates keyword-based search queries. Note that both Apoidea and the distributed indexing system can

share a common underlying lookup protocol layer based on DHT-based P2P systems. Further, one can package the Apoidea and the distributed indexing scheme as a screen saver (like SETI@home [15]) so that the search engine on each node uses its idle CPU cycles and network bandwidth.

6 Related Work

Most of the research in the area of web crawlers has been based on centralized architectures. While [5,10,6] were a single machine architectures, [16] presents the design of a distributed crawler. The common feature of all those architectures being a centralized location storing all the critical data structures and a central manager controlling various crawler components. Such an architecture is complicated and requires high maintenance. Since the controlling organization will be using a single link with the internet, it requires a very high bandwidth connection. In addition, it requires complicated load balancing for HTTP and DNS requests.

The P2P decentralized solutions, in addition to not suffering from above mentioned issues has other potential benefits as well. Having an autonomous solution prevents costly administration requirements and is more fault tolerant. Also it has the immense potential to exploit geographical proximity of various peers to the web resources. There have been a few attempts at developing P2P decentralized versions of crawlers. Boldi et al [4] looked at the fault tolerance properties of such crawlers, showing that such an architecture will be more fault tolerant. [18] also attempted to provide a loosely connected P2P design. However, both of those works did not provide a concrete mechanism of assigning jobs to various peers and load balancing between them.

We would also like to mention the work done in the area of distributed P2P lookup protocols. There are several P2P protocols proposed so far [2,13,14,17]. Apoidea P2P web crawler is built on top of Chord [17]. Most of the routed query based P2P protocols are motivated by file sharing applications. However, none of them are really suitable for file sharing applications due to the problems in providing flexible mappings from file name search phrases to file identifiers. (One such insufficient mapping is provided in [1]). It is interesting to see that web crawling can benefit from a P2P approach, being an extremely suitable application domain for routed query based P2P protocols.

7 Conclusions and Future Work

In this paper, we have presented a decentralized P2P architecture for a web crawler. We have described the complete design and structure of the system. We show how optimizations based on geographical proximity of peers to the web resources can significantly better the crawling performance and results. We have also presented a sample mapping scheme that achieves this goal. Also, through our initial experiments we have shown the nice scalability of our architecture.

For the future, we would like to run the system in a wide scenario and test its properties. On the research side, we would like to enhance the URL mapping scheme in a way that implicitly handles geographical proximity. Also, security is a key problem when using an autonomous collection of peers. We intend to explore issues preventing malicious behavior in the network.

References

1. K. Aberer. P-grid: A self-organizing access structure for p2p information systems. In *Sixth International Conference on Cooperative Information Systems*, 2001.
2. J. Kubaitowics B. Zhao and A. Joseph. Tapestry: An infrastructure for fault-tolerance wide-area location and routing. Technical Report UCB/CSD-01-1141, University of California, Berkeley, 2001.
3. Burton Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.
4. P. Boldi, B. Codenotti, M. Santini, and S. Vigna. Ubicrawler: Scalability and fault-tolerance issues. In *Poster Proceedings of 11th International World Wide Web Conference.*, 2002.
5. 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.
6. M. Burner. Crawling towards eternity: Building an archive of the world wide web. In *Web Techniques*, 1997.
7. Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009:46+, 2001.
8. Omprakash D. Gnawali. A keyword-set search system for peer-to-peer networks.
9. Gnutella. The gnutella home page. http://gnutella.wego.com/, 2002.
10. Allan Heydon and Marc Najork. Mercator: A scalable, extensible web crawler. *World Wide Web*, 2(4):219–229, 1999.
11. Kazaa. The kazaa home page. http://www.kazaa.com/, 2002.
12. T. Lu, S. Sinha, and A. Sudam. Panache: A scalable distributed index for keyword search. Technical report, 2002.
13. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. 2001.
14. A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, November 2001.
15. SETI@home. The seti@home home page. http://setiathome.ssl.berkeley.edu.
16. Vladislav Shkapenyuk and Torsten Suel. Design and implementation of a high-performance distributed web crawler. In *Proceedings of International Conference on Data Engineering*, 2002.
17. I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of SIGCOMM Annual Conference on Data Communication*, August 2001.
18. T. Takahashi, H. Soonsang, K. Taura, and A. Tonezawa. World wide web crawler. In *Poster Proceedings of 11th International World Wide Web Conference.*, 2002.