Design and Implementation of Scalable, Fully Distributed Web Crawler for a Web Search Engine

M. Sunil Kumar
Associate Professor
C R Engineering College, Tirupati
Research Scholar
S V University
Tirupati

P.Neelima
Assistant Professor
C R Engineering College, Tirupati
Tirupati

ABSTRACT

The Web is a context in which traditional Information Retrieval methods are challenged. Given the volume of the Web and its speed of change, the coverage of modern web search engines is relatively small. Search engines attempt to crawl the web exhaustively with crawler for new pages, and to keep track of changes made to pages visited earlier. The centralized design of crawlers introduces limitations in the design of search engines. It has been recognized that as the size of the web grows, it is imperative to parallelize the crawling process. Contents other then standard documents (Multimedia content and Databases etc) also makes searching harder since these contents are not visible to the traditional crawlers. Most of the sites stores and retrieves data from backend databases which are not accessible to the crawlers. This results in the problem of hidden web. This paper proposes and implements DCrawler, a scalable, fully distributed web crawler. The main features of this crawler are platform independence, decentralization of tasks, a very effective assignment function for partitioning the domain to crawl, and the ability to cooperate with web servers. By improving the cooperation between web server and crawler, the most recent and updates results can be obtained from the search engine. A new model and architecture for a Web crawler that tightly integrates the crawler with the rest of the search engine is designed first. The development and implementation are discussed in detail. Simple tests with distributed web crawlers successfully show that the Dcrawler performs better then traditional centralized crawlers. The mutual performance gain increases as more crawlers are added.

Keywords: web search engines, crawling, software architecture

1. INTRODUCTION:

1.1 Web search engines and crawlers

1.1.1 Information Retrieval (IR)

Information Retrieval is the area of computer science concerned with retrieving information about a subject from a collection of data objects. This is not the same as Data Retrieval, which in the context of documents consists mainly in determining which documents of a collection contain the keywords of a user query. Information Retrieval deals with satisfying a user need. Although there was an important body of Information Retrieval techniques published before the invention of the World Wide Web, here are unique characteristics of the Web that made them unsuitable or insufficient.

The low cost of publishing in the "open Web" is a key part of its success, but implies that searching information on the Web will always be inherently more difficult then searching information in traditional, closed repositories.

1.1.2 Web search and Web crawling

The typical design of search engines is a "cascade", in which a Web crawler creates a collection which is indexed and searched. Most of the designs of search engines consider the Web crawler as just a first stage in Web search, with little feedback from the ranking algorithms to the crawling process. This is a cascade model, in which operations are executed in strict order: first crawling, then indexing, and then searching.

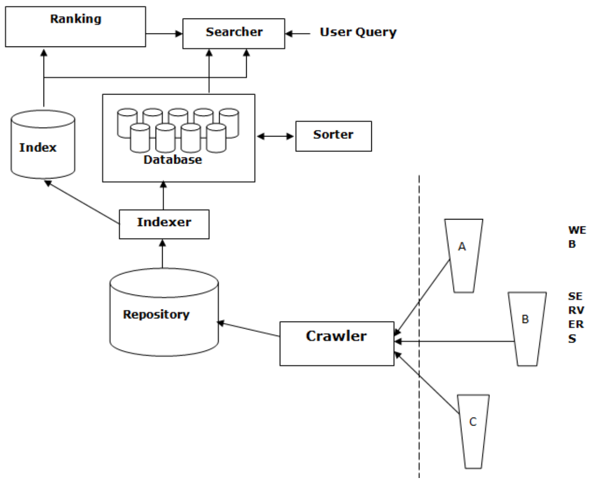

Figure 0.1 Structure of Web search Engine

An aim of this approach is to provide the crawler with access to all the information about the collection to guide the crawling process effectively. This can be taken one step further, as there are tools available for dealing with all the possible interactions between the modules of a search engine,

1.2 Working of a crawler

A web crawler is an automatic web object retrieval system that exploits the web's dense link structure. It has two primary goals, to seek out new web objects, and to observe changes in previously-discovered web objects (web-event detection).The basic web crawler algorithm has not changed since the World Wide Web Wanderer (the first reported web crawler) was designed in 1993. Almost all crawlers follow some variant of the basic web-traversal algorithm. Crawlers must continue to deal with issues of scalability as the World-Wide Web expands. How does one efficiently and effectively crawl the current set of almost 2.5 billion publicly index-able web pages if crawlers are limited by crawling speed and difficulty in predicting web-events? The speed at which a crawler can traverse the web is limited by a number of factors, including the bandwidth of the crawler and the latency of the network.

Predicting when a web object is going to change, helps to limit the amount of useless polling done by a crawler to determine if it has been updated since the last visit. The fewer resources wasted by a crawler doing useless polls, the more that can be delegated to the task of locating new information. In the end, crawlers are going to be relying upon communicating with others and being instances of themselves (in the parallel sense), This arises the need for autonomously cooperative sharing web crawlers - crawlers that can make decisions on their own, and communicate with others when the need arises.

2.REQUIRMENTS AND ASSUMPTIONS

It has been recognized that as the size of the web grows, it becomes imperative to parallelize the crawling process, in order to finish downloading pages in a reasonable amount of time (Junghoo 2002). Nonetheless, little published work actually investigates the fundamental issues underlying the parallelization of the different tasks involved with the crawling process (Sergey 1998, Alan 2001). For example, some features of Google have been presented in (Sergey 1998), where the crawling mechanism is described as a two stage process: a URL server distributes individual URLs to multiple crawlers, which download web pages in parallel; the crawlers then send the downloaded pages to a central indexer, on which links are extracted and sent via the URL server to the crawlers.

In contrast, when designing DCrawler, all the tasks of web crawlers were decentralized, with obvious advantages in terms of scalability and fault tolerance.

Essential features of DCrawler are

- Platform independence;
- Full distribution of every task
- Tolerance to failures:
- Scalability.

Following sections describes the design goals and assumptions which have guided the architectural choices of DCrawler. These features are the offspring of a well defined design goal: fault tolerance and full distribution (lack of any centralized control). For instance, while there are several reasonable ways to partition the domain to be crawled if one assume the presence of a central server, it becomes harder to find an assignment of URLs to different agents which is fully distributed, does not require too much coordination, and allow to cope with failures.

2.1 Design Goals

2.1.1 Full Distribution

In order to achieve significant advantages in terms of programming, deployment, and debugging, a parallel and distributed crawler should be composed by identically programmed agents, distinguished by a unique identifier only. This has a fundamental consequence: each task must be performed in a fully distributed fashion, that is, no central coordinator can exist.

Also no assumption concerning the location of the agents is made, and this implies that latency can become and issue, so that communication should be minimized to reduce it.

2.1.2 Balanced locally computable assignment.

The distribution of URLs to agents is an important issue, crucially related to the efficiency of the distributed crawling process.

Following goals three goals are identified as important:

- At any time, each URL should be assigned to a specific agent, which is solely *responsible* for it.
- For any given URL, the knowledge of its responsible agent should be locally available. In other words, every agent should have the capability to compute the identifier of the agent responsible for a URL, without communicating.
- The distribution of URLs should be *balanced*, that is, each agent should be responsible for approximately the same number of URLs.
-

2.1.3 Scalability

The number of pages crawled per second per agent should be (almost) independent of the number of agents. In other words, the throughput should grow linearly with the number of agents.

2.1.4 Platform independence

The crawler should be able to work among different platforms and in heterogeneous networks with different architecture. This is the reason for choosing java for implementation of Drawler.

2.1.5 Cooperation with web server

The crawler should have the ability to cooperate with the web server during the crawling process. The network traffic generated by the crawlers can be considerably reduced using this mechanism.

2.1.6 Politeness

A parallel crawler should never try to fetch more than one page at a time from a given host. Congestion that could arise because of multiple threads crawling single host could be avoided with this technique.

2.1.7 Fault tolerance

A distributed crawler should continue to work under *crash faults*, that is, when some agents abruptly die. No behavior can be assumed in the presence of this kind of crash, except that the faulty agent stops communicating; in particular, one cannot prescribe any action to a crashing agent, or recover its state afterwards (note that this is different from milder assumptions, as for instance saying that the state of a faulty agent can be recovered. In the latter case, one can try to "mend" the crawler's global state by analyzing the state of the crashed agent). When an agent crashes, the remaining agents should continue to satisfy the "Balanced locally computable assignment" requirement: this means, in particular, that URLs will have to be redistributed.

This has two important consequences:

- It is not possible to assume that URLs are statically distributed.
- Since the "Balanced locally computable assignment" requirement must be satisfied *at any time*, it is not reasonable to rely on a distributed reassignment protocol after a crash. Indeed, during the protocol the requirement would be violated.

3. THE BASIC DESIGN

3.1.1 Multithread architecture

DCrawler is composed of several agents that autonomously coordinate their behavior in such a way that each of them scans its share of the web. An agent performs its task by running several threads, each dedicated to the visit of a single host. More precisely, each thread scans a single host using a breadth-first visit.

Several mechanisms are used to make sure that different threads visit different hosts at the same time, so that each host is not overloaded by too many requests. The out links that are not local to the given host are dispatched to the right agent, which puts them in the queue of pages to be visited. Thus, the overall visit of the web is breadth first, but as soon as a new host is met, it is entirely visited (possibly with bounds on the depth reached or on the overall number of pages), again in a breadth-first fashion.

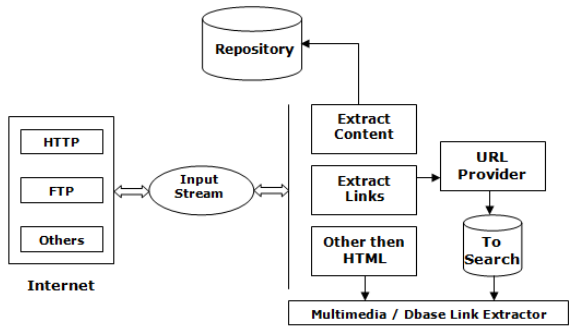

Figure 2.1.1.1 Single Thread in Crawler

3.1.2 Quality of Pages

More sophisticated approaches (which can take into account suitable priorities related to URLs, such as, for instance, their rank) can be easily implemented. However it is worth noting that several authors have argued that breadth-first visits tend to find high quality pages early on in the crawl.

An important advantage of per-host breadth-first visits is that DNS requests are infrequent. Web crawlers that use a global breadth-first strategy must work around the high latency of DNS servers: this is usually obtained by buffering requests through a multithreaded cache. Similarly, no caching is needed for the robots . txt file required by the "Robot Exclusion Standard" indeed such file can be downloaded any time an host breadth-first visit begins.

3.1.3 Assignment of URLs

Assignment of hosts to agents takes into accounts the mass storage resources and bandwidth available at each agent. This is currently done by means of a single indicator, called capacity, which acts as a weight used by the assignment function to distribute hosts. Under certain circumstances, each agent a gets a fraction of hosts proportional to its capacity Ca. Note that even if the number of URLs per host varies wildly, the distribution of URLs among agents tends to even out during large crawls. Besides empirical statistical reasons for this, there are also other motivations, such as the usage of policies for bounding the maximum number of pages crawled from a host and the maximum depth of a visit. Such policies are necessary to avoid (possibly malicious) web traps.

3.1.4 Failure Detector

Finally, an essential component in DCrawler is a reliable failure detector that uses timeouts to detect crashed agents; reliability refers to the fact that a crashed agent will eventually be distrusted by every active agent (a property that is usually referred to as strong completeness in the theory of failure detectors). The failure detector is the only synchronous component of DCrawler (i.e., the only component using timings for its functioning); all other components interact in a completely asynchronous way

3.2 Software Architecture

3.2.1 The overall structure

DCrawler is composed by several agents that autonomously coordinate their behavior in such a way that each of them scans its share of the web. The objective of the design of this crawling architecture is to divide the crawling task into different tasks that will be carried efficiently by specialized modules.

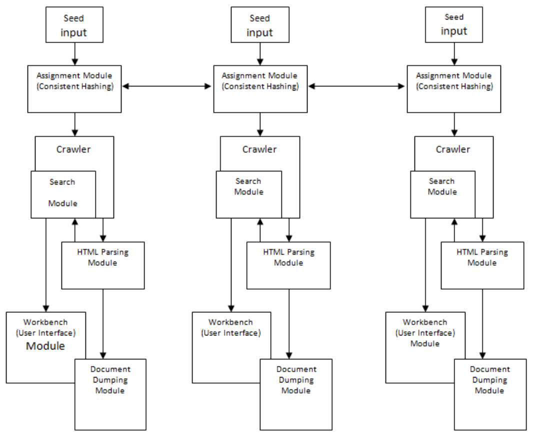

Figure 2.1.1.2 Multiple agents cooperation

Figure 3.2 shows three independent crawlers, and their cooperation logic. The seed is the starting URL list provided to the crawler. The crawler starts from seed URLs. The assignment module process each and every URL crawler come across and locally computes the crawler responsible for specific host (which is done using identifier based consistent hashing as explained in next chapter). The URL is then given to core crawling module.

3.2.2 The core Crawling Module

The core crawling module follows the very basic crawling algorithm. The page related to the corresponding URL is fetched first, which is then passed to the HTML parsing module. The HTML parsing module extracts different components of the web page and returns them to crawler in turn. The extracted links are passed to assignment module for further processing.

3.2.3 HTML Parsing

HTML parsing module analyses the web pages fetched by the core crawling module. Apart from link extraction, the HTML module can be used by the search engine directly to search for specific keywords. This approach of crawling for a specified set of pages based on constrains is known as focused crawling.

3.2.4 Workbench Interface

The workbench is a user interface module that provides graphical representation and statistics on the current crawling process. This also allows to add seeds during crawler execution and to control crawling process in real time.

3.2.5 Document Dumping

When a page is fetched, after link extraction it should be stored in the repository. This is done using document dumping module. The location can be in local machine or network server. When the dumping is done in local machine, then available capacity is also considered as a component that forms the weight of the crawler.

4. WEB SERVER COOPERATION SCHEMES

4.1 The assignment functions

4.1.1 Required assignment function

Let *A* be the set of agent identifiers (i.e., potential agent names), and *L* be the subset of living agents: Then the assignment function have to assign hosts to agents in *L*. More precisely, a function δ need to be set up in such a way that, for each nonempty set *L* of alive agent, and for each host *h*, delegates the responsibility of fetching *h* to the agent $\delta_L(h)$ in *L*. The following properties are desirable for an assignment function:

***Balancing**:* Each agent should get approximately the same number of hosts; in other words, if *m* is the (total) number of hosts,

***Contravariance**:* The set of hosts assigned to an agent should change in a contravariant manner with respect to the set of alive agents across a deactivation and reactivation, that is to say, if the number of agents grows, the portion of the web crawled by each agent must shrink.

Contravariance has a fundamental consequence: if a new set of agents is added, no old agent will ever lose an assignment in favor of another old agent. This guarantees that at any time the set of agents can be enlarged with minimal interference with the current host assignment.

4.1.2 Existing approaches

Satisfying partially the above requirement is not difficult: for instance, a typical approach used in non-fault-tolerant distributed crawlers is to compute a modulo-based hash function of the host name. This has very good balancing properties (each agent gets approximately the same number of hosts), and certainly can be computed locally by each agent knowing just the set of alive agents.

But when an agent crashes, the assignment function need to be computed again, giving however a different result for almost all hosts. The *size* of the sets of hosts assigned to each agent would grow or shrink contravariantly, but the *content* of those sets would change in a completely chaotic way. As a consequence, after a crash most pages will be stored by an agent that should not have fetched them, and they could mistakenly be re-fetched several times (for the same reason, a modulo-based hash function would make it difficult to increase the number of agents during a crawl).

Clearly, if a central coordinator is available or if the agents can engage a kind of "resynchronization phase" they could gather other information and use other mechanisms to redistribute the hosts to crawl. However, shifting the fault-tolerance problem to the resynchronization phase-faults in the latter would be fatal.

4.1.3 Background

Although it is not completely obvious, it is not difficult to show that contravariance implies that each possible host induces a total order (i.e., a permutation) on *A*; more precisely, a contravariant assignment is equivalent to a function that assigns an element of S_A (the symmetric group over *A*, i.e., the set of all permutations elements of *A*, or equivalently, the set of all total orderings of elements of *A*) to each host: then, $\delta_L(h)$ is computed by taking, in the permutation associated to *h*, the first agent that belongs to the set *L*.

A simple technique to obtain a balanced, contravariant assignment function consists in trying to generate such permutations, for instance, using some bits extracted from a host name to seed a (pseudo)random generator, and then permuting randomly the set of possible agents. This solution has the big disadvantage of running in time and space proportional to the set of possible agents (which one wants to keep as large as feasible). Thus, a more sophisticated approach is needed.

4.1.4 Consistent Hashing

Recently, a new type of hashing called *consistent hashing* (David 1997, David 1999) has been proposed for the implementation of a system of distributed web caches (a different approach to the same problem can be found in (Robert 1993)). The idea of consistent hashing is very simple, yet profound.

For a typical hash function, adding a bucket (i.e., a new place in the hash table) is a catastrophic event. In consistent hashing, instead, each bucket is replicated a fixed number *k* of times, and each copy (called as a *replica*) is mapped randomly on the unit circle. When key needs to be hashed, points in the unit circle is computed in some way from the key, and find its nearest replica: the corresponding bucket is the required hash. Consistent hashing, which in particular gives balancing for free. Contravariance is also easily verified.

In this case, buckets are agents, and keys are hosts. However, if contravariance need to hold, care should be taken, because mapping randomly the replicas to the unit circle each time an agent is started will not work; indeed, δ would depend not only on *L*, but also on the choice of the replicas. Thus, *all* agents should compute the same set of replicas corresponding to a given agent, so that, once a host is turned into a point of the unit circle, all agents will agree on who is responsible for that host.

4.1.5 Identifier-Seeded Consistent Hashing

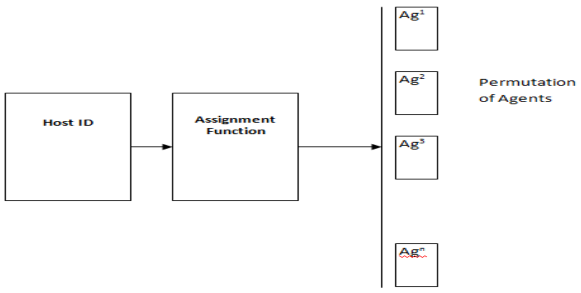

Figure 2.1.1.3 Identifier seeded consistent hashing

A method to fix the set of replicas associated to an agent and try to maintain the good randomness properties of consistent hashing is to derive the set of replicas from a very good random number generator seeded with the agent identifier: this approach is called as *identifier-seeded consistent hashing*. In the implementation, for the java Random number generator library is used, which is a fast random generator.

When a new agent is started, its identifier is used to generate the replicas for the agent. However, if during this process a replica that is already assigned to some other agent is generated; the new agent must be forced to choose another identifier.

This solution might be a source of problems if an agent goes down for a while and discovers a conflict when it is restarted. Nonetheless, some standard probability arguments show that with a 64-bit representation for the elements of the unit circle there is room for 10^4 agents with a conflict probability of 10^{-12}.

Theoretical analysis of the balancing produced by identifier-seeded consistent hashing is most difficult, if not impossible (unless, of course, one uses the working assumption that replicas behave as if randomly distributed). Thus, experimental data is reported (Paolo 2001). In Figure 4.2 it is clear that once a substantial number of hosts have been crawled, the deviation from perfect balancing is less than 6% for small as well as for large sets of agents when $\kappa = 100$, that is, for 100 replicas per bucket; if $\kappa = 200$, the deviation decreases to 4.5%.

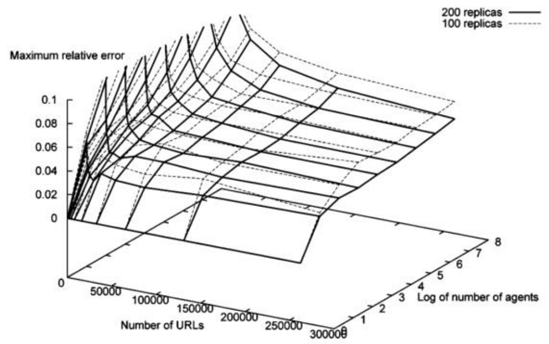

Figure 2.1.1.4 Experimental data on identifier-seeded consistent hashing

4.2 Web Server Coordination

The standard HTTP transaction in which a web crawler fetches a page from a web server is shown in Figure 4.3. Note that metadata (information about the page) is downloaded along with the data.

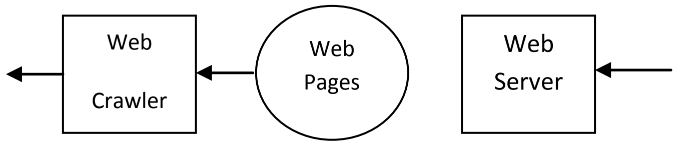

Figure 2.1.1.5 Standard HTTP Transaction

The cooperation schemes can be divided in two main groups: *interrupt* and *polling. A* crawler may use one of them or a combination of different schemas. In the ***interrupt*** **(or *push)*** schemes, the web server begins a transaction with the search engine whenever it is necessary. This is similar to the relationship between the main processor and a hardware device (network card, scanner, etc.) in a modern computer. In *the* ***polling*** **(or *pull)*** schemes, the search engine periodically requests data from the web server, based on search engine policies.

For DCrawler a different cooperation scheme is implemented from above, that utilizes both push and pull schemes.The interface which is basically designed to fetch HTML pages can be extended so that it may allow agents to fetch contents other then HTML in secure way. With this kind of interface two major problems in web crawling can be solved.

- A secure interface between Database/Multimedia content on the server and agent can be created. The problem of invisible web/ Hidden web can be solved with this interface.
- The server may cache the recently searched pages and search terms. This will improve the performance during crawling process and the collaboration between different users searching in the web can be achieved.

IV. CONCLUSION

Web crawling was described in the context of information retrieval. Although there are many studies about web search, web crawler designs and algorithms, they are mostly kept as business secrets, with some exceptions.

DCrawler, a fully distributed, scalable and fault-tolerant web crawler was presented. This project dealt with web crawling from a practical point of view. From this point of view, a series of problems were faced during the design and implementation of a Web crawler. DCrawler introduces new ideas in parallel crawling, and in the domain of crawler – web server coordination. With consistent hashing, graceful degradation in the presence of faults and linear scalability are made possible. The development and deployment of DCrawler will be continued and the performance will be tested in very large web domains.

This Web crawler can efficiently download several million pages per day, and can be used for web search and web characterization. Several web portals have been downloaded and analyzed using DCrawler, extracting statistics on HTML pages, multimedia files, Web sites and link structure.

V. REFERENCES

[1] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. (1997) Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In Proc. of the 29th Annual ACM Symposium on Theory of Computing, pages 654-663, El Paso, Texas,

[2] David Karger, Tom Leighton, Danny Lewin, and Alex Sherman. (1999.) Web caching with consistent hashing. In Proc. of 8th International World-Wide Web Conference, Toronto, Canada.

[3] Demetrios Zeinalipour-Yazti and Marios Dikaiakos. (2002) Design and implementation of a distributed crawler and filtering processor. In Proc. ofNGITS 2002, volume 2382 of Lecture Notes in Computer Science, pages 58-74.

[4] Hongfei Yan, Jianyong Wang, Xiaoming Li, and Lin Guo. (2002) Architectural design and evaluation of an efficient

Web-crawling system. The Journal of Systems and Software, 60(3): 185-193,.

[5] Java™ remote method invocation (RMI). http: //Java . sun.com/products/jdk/rmi/.

[6] Marc Najork and Janet L. Wiener. (2001) Breadth-first search crawling yields high-quality pages. In Proc. of 10th International World Wide Web Conference, Hong Kong, China,.

[7] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. (2001) Trova-tore: Towards a highly scalable distributed web crawler. In Poster Proc. of Tenth International World Wide Web Conference, pages 140-141, Hong Kong, China,

[8] Robert Devine. (1993) Design and implementation of DDH: A distributed dynamic hashing algorithm. In David B. Lomet, editor, Proc. Foundations of Data Organization and Algorithms, 4th International Conference, FODO'93, volume 730 of Lecture Notes in Computer Science, pages 101-114, Chicago, Illinois, USA,. Springer-Verlag.

[9] Tushar Deepak Chandra and Sam Toueg. (1996) Unreliable failure detectors for reliable distributed systems. Journal of the ACM, 43(2):225-267,.

[10] Vladislav Shkapenyuk and Torsten Suel. (2002) Design and implementation of a high-performance distributed web crawler. In IEEE International Conference on Data Engineering (ICDE),.

[11] L. Lamport, “Paxos made simple,” ACM SIGACT

[12] *News*, vol. 32, no. 4, pp. 51–58, December 2001.

[13] V. Paxson, “End-to-end routing behavior in the

[14] Internet,”*ACM SICOMM Computer Communication Re- view*, vol. 35, no. 5, pp. 43–56, October 2006.

[15] A. Crespo and H. Garcia-Molina, “Semantic

[16] overlaynetworks for p2p systems,” Stanford University, Tech. Rep. 2003-75, 2003.

[17] M. Bender, S. Michel, P. Triantafillou, G.

[18] Weikum, and C. Zimmer, “P2p content search: Give the Web back to the people,” February 2006, international Workshop on Peer-to-Peer Systems (IPTPS).

[19] M. Shokouhi, J. Zobel, F. Scholer, and S. Tahaghoghi,“Capturing collection size for distributed noncooperative retrieval,” in *Proceedings of the Annual ACM SIGIR Conference*. Seattle, WA, USA: ACM

[20] Press, August 2006.

[21] L. A. Barroso, J. Dean, and U. H¨olzle, “Web search for a planet: The Google Cluster Architecture,” *IEEE Micro*, vol. 23, no. 2, pp. 22–28,Mar./Apr. 2003.

[22] A.-J. Su, D. Choffnes, A. Kuzmanovic, and F. Bustamante, “Drafting behind Akamai travelocity-based detouring),” in *Proceedings of the ACM SIGCOMM Conference*, Pisa, Italy, September 2006, pp. 435–446.

[23] S. M. Beitzel, E. C. Jensen, A. Chowdhury, D. Grossman, and O. Frieder, “Hourly analysis of a very large topically categorized web query log,” in *SIGIR ’04: Proceedings of the 27th annual international ACM SI- GIR conference on Research and development in in- formation retrieval*. New York, NY, USA: ACM Press, 2004, pp. 321–328.

[24] K. Risvik and R. Michelsen, “Search engines and web dynamics,” *Computer Networks*, pp. 289–302, 2002.

[25] F. Cacheda, V. Carneiro, V. Plachouras, and I. Ounis, “Performance analysis of distributed information retrieval architectures using an improved network simulation model,” *Information Processing and Management*, vol. 43, no. 1, pp. 204–224, 2007.

[26] W. B. Cavnar and J. M. Trenkle, “N-gram-based text categorization,” in *Proceedings of SDAIR-94, 3rd An- nual Symposium on Document Analysis and Informa- tion Retrieval*, Las Vegas, US, 1994, pp. 161–175.

[27] G. Grefenstette, “Comparing two language identification schemes,” in *Proceedings of the 3rd interna- tional conference on Statistical Analysis of Textual Data (JADT 1995)*, 1995.

V. ABOUT AUTHORS

Mr. M Sunil Kumar has completed B.Tech in Computer Science & Information Technology from JNT University and M.Tech in Computer Science from JNT University. Presently he is pursuing Ph.D in Computer Science and Engineering, S.V.University, TIRUPATI. He is currently working as Associate Professor in the Department of CSE, Chadalwada Ramanamma Engineering College, Tirupati, A.P. His main research interest includes Software Engineering, Software Architecture, Information Retrieval and Database Management Systems.

P.Neelima has completed B.Tech in Computer Science and Engineering from JNT University.She is currently working as Assistsnt Professor in the Department of CSE, Chadalwada Ramanamma Engineering College, Tirupati, A.P. Her main research interest includes Software Engineering, Software Architecture, Information Retrieval and Database Management Systems.