**DistSearch**

**Brad Westhafer**

**COMP 512**

**Spring 2018**

**Table of Contents**

Introduction……………………………………………………………...3

Related Work………………………..…………………………………...3

Challenges and Pitfalls……………..……………………………………4

Implementation…………………………………………………………. 5

Performance Evaluation…………………………………………………7

Conclusion and Future Work…………………………………………….8

References……………………………………………………………….8

**Introduction**

DistSearch is a web search engine that maintains a peer to peer distributed index. It is written in C++ and utilizes Boost libraries for fetching web pages and passing messages. The index is stored in separate SQLite databases at each peer. A central server maintains a list of peers and can send messages to them or order them to terminate.

A distributed system has significant advantages for web search. According to Google, there were over 130 trillion pages on the web in November 2016[10]. A year and a half later, this number is likely even larger and will probably continue to grow. This is far too large to realistically index with a centralized approach. The largest hard drive in the world right now is a 60 TB solid state drive[1]. Given that 1 terabyte is approximately 1 trillion bytes, the total size of this hard drive is approximately 60 trillion bytes. Even assuming one could index each page and its uniform resource locator (URL) in just 1 kilobyte of space, one would only be able to index 60 million web pages on this hard drive. By distributing the index across many smaller hard drives, a distributed search engine can greatly reduce the cost of launching a new search engine.

The motivation behind this project was to create a new search engine that would be resistant to censorship and biased search results while respecting the privacy of its users. Major corporate search engines have faced concerns over privacy [12], political bias in their result ranking [11], acting as gatekeepers [5] and “filter bubbles” that attempt to bias results toward whatever they think users want to see instead of providing everybody with the same set of search results[9]. Given the advantages of a distributed system, the author felt that the topic of a distributed search engine was worth exploring.

**Related Work**

Faroo and Yacy are existing peer-to-peer distributed search engines. Faroo indexes and ranks pages based upon the web traffic of their peer machines[3]. This approach has its merits, as AV-Test found significantly less malware in Faroo search results compared to search results from Google, Bing and Yandex[2]. However, we can see the disadvantage of this approach by searching Faroo for information on such specialized topics as Dijkstra’s algorithm and distributed deadlock detection.

Yacy takes a different approach, with each peer controlling their own indexing, including the ability to rank results as they wish[6]. While such an approach is resistant to censorship, it is not resistant to filter bubbles. In fact, by design it permits users to create and control their own filter bubble.

**Challenges and Pitfalls**

Over the course of this project, the author faced a number of challenges and pitfalls that required the project to be scaled down compared to the original proposal. The initial plan had been to develop a GUI within the first 2 weeks of the project, followed by 8 weeks of development on the peer-to-peer distributed search engine and 2 weeks to write this paper. However, due to delays related to getting necessary software installed for the GUI framework, the author decided to abandon work on the GUI to focus on the underlying distributed web crawler.

After abandoning the plan to include a GUI, the learning curve for using MPI, the message passing library named in the proposal, was longer than expected and involved difficulties in setting up MPI to use the range of ports required to get around a firewall. While the firewall issue was eventually resolved, it slowed down progress on the project by several weeks.

The last big issue that stood in the way of starting the project was finding a way to get the contents of web pages. Ultimately, the decision was made to go with Boost.Asio for getting web pages. While this worked for web pages passed under the http protocol, this did not work with web pages passed over the more secure https protocol that most web pages have recently updated to. Unfortunately, the English Wikipedia, originally planned to be the content source for this project, is now an https-only site. While some effort was made to modify this code for compatibility with https, time was rapidly running out so these efforts were abandoned in favor of just indexing http compatible sites.

Once development actually started, there were two major decisions that needed to be made. The first decision involved how to store the index. The original plan had involved implementing some sort of index file but, in the interest of finishing the project, that was abandoned in favor of using SQLite. The other major design issue involved finding a way to quickly determine if two versions of a URL were exactly the same. This was eventually solved through the use of the MD5 hashing algorithm. While MD5 is a poor algorithm for cryptography because it is trivially easy to break, it was suitable for the purpose for which it is being used in this project. I also had to decide on a method of timestamping URLs. Ultimately, I decided to use the system’s local epoch time. While this does not guarantee that peer A indexed some URL before peer B in real time, this does not matter for the purposes of this project as the timestamp only comes into play to remove duplicates.

**Implementation**

DistSearch is implemented in C++ using Boost.MPI for message passing, Boost.Asio for fetching URLs, SQLite for storing the distributed indexes and MD5 for hashing. In MPI, the n processes are numbered from 0 to n – 1. In DistSearch, process 0 is a search box process that has the responsibility of receiving search queries from the user, broadcasting them to the other n – 1 processes and waiting for them to return the results. Once it receives results, it displays them. Process 0 can also send termination orders to any other process and will remember which processes it has terminated.

The other n – 1 processes each maintain a separate queue of URLs to index and a separate index of URLs. Upon starting DistSearch, they restore their respective queues. Then, they dequeue a URL from the queue, add it to their visited list and fetch the URL from the web server. If the web server is down, they place it at the back of the queue and move onto the next URL.

If, however, the web server is working, they download the URL and parse its contents searching for links. Any linked URLs are added to the queue, if they are not already in the queue and not in the visited list. Then, every peer messages every other peer to check if the URL is a duplicate, including a timestamp in the message. While waiting for responses, each peer adds their URL to their respective indexes. If a peer determines that they have a URL, then that peer checks the respective timestamps. If their timestamp is earlier or the timestamps are equal and the other peer has a lower ID number, then they send a return message containing the URL and the MD5 hash of the contents of their version of the URL to the other peer. Upon receipt of a message with a URL followed by a hash, a peer fetches the URL from its database and checks if the hashes match. If the hashes match, then the peer deletes the URL. Otherwise, the peer sends a message to the other peer ordering the other peer to delete the URL from its index.

Messages are sent using Boost.MPI’s isend function, which is a non-blocking send. When all processes check for a message, they use Boost.MPI’s iprobe function, which is a non-blocking probe. If the probe says that a message is available, then the process uses the recv function of Boost.MPI, which is a blocking receive, to receive the message. The only exception to this rule occurs when process 0 decides to search all peers simultaneously. In this case, process 0 sends a blank message to all of the peers then uses Boost.MPI’s broadcast function to send a message containing a search query to all of the other peers. Once the peers discover that such a message is out there, they receive the broadcast from process 0 using a Boost.MPI broadcast function call of their own.

On receipt of a broadcast message, the peers query their database for the search term. Then, they return results to process 0 or, if no results are found, they return “no results” instead. They conclude the process of processing a search query by calling Boost.MPI’s recv to “receive” the blank message. This prevents peers from erroneously running the broadcast function call when there is no search query to receive. In one version of the program, peers called broadcast on every single iteration, which caused them to block until they received a message and, in the process, also prevented the program from terminating. The blank messages were introduced to address this bug.

The termination condition for the peers is set to whenever their queue is empty or they receive a termination order from process 0. Process 0 terminates whenever it has sent a termination order to all peers. While terminating, the peers save the contents of their respective queues to a file but do not save their visited lists. This permits peers to revisit previously visited pages on future runs to keep them up to date. The decision to adopt this approach was to avoid two extreme scenarios, both of which are undesirable. In the first scenario, a peer repeatedly indexes the same page. If this happens, then the peer is doing unnecessary work and wasting resources that should be used to index more pages. This would be possible if the peer did not keep a visited list at all. The other scenario involves a peer never revisiting a page it has already visited because it keeps a permanent visited list. Besides the wasted disc space involved in indefinitely storing a list of visited URLs, this would be very bad because the contents of web pages change over time, so large portions of the distributed index could end up out of date because the only peer that indexed them would be unable to find them! Thus, the decision was made to use visited lists but discard them at the conclusion of each session in order to achieve a middle ground between the two undesirable extremes.

**Performance Evaluation**

While the final result of this project is not, in any sense, a viable web search engine that is ready to compete with the existing distributed search engines, this project provides a proof of concept and confirms that a distributed approach is potentially viable. Thus, DistSearch is a foundation upon which future work in this area can be built.

In its current form, DistSearch merely returns a list of URLs from its distributed index in no particular order. Many of the URLs returned appear to contain identical or virtually identical content. These issues are due to the lack of a ranking algorithm of any kind and the lack of any method for recognizing duplicate URLs.

Another issue for DistSearch in its current form is the lack of any way to limit the size of a peer’s index. If this project were left running idle for a lengthy period of time, this could potentially consume the entire hard disk space on peer machines.

The biggest performance issue however is that the current project architecture would likely have scalability problems. Besides the requirement to utilize the mpirun command on a single machine to launch all processes on all machines simultaneously, querying all peers simultaneously is unlikely to scale well. It is hard to imagine receiving this approach working if there were a million peers instead of 11.

Ultimately, the current version of DistSearch is unlikely to work outside of the specific environment in which it was developed. Within said environment, DistSearch runs and generally returns results fairly quickly but is not yet a viable alternative to existing search engines and is sometimes prone to slowdowns if a peer is in the middle of parsing a large page when a search query comes in or the network slows down. If one of the peers crashes without the knowledge of the root, this could freeze the entire system if the root attempts to search while this peer is down since it will wait forever for results that will never arrive. These issues will need to be addressed in the future.

**Conclusion and Future Work**

While DistSearch is a proof of concept for a distributed search engine, future work will be needed before it is a viable option for web search. There are a number of areas where this work can be improved upon. These include the development of a GUI, exploration of various ranking algorithms, replacement of MPI with a message passing system that is better suited to this project as originally envisioned and research into ways to detect duplicate or near-duplicate pages with different URLs to save resources. In addition, some way to set limits for the sizes of the various indexes would be desirable.

Of those areas to explore, the ones most directly related to web search are the ranking algorithm and removing duplicate URLs. For ranking algorithms, Google’s PageRank algorithm [8] is a good starting point for research. The duplicate URL problem has also been covered in the literature [4] and would be a good area to explore in future work.

Ultimately, this project did not accomplish the ambitious goals that were set out in the proposal three months ago but it was a starting point and a good learning experience for developing search engines and for developing distributed systems.

**References**

1: Anthony, S. Seagate’s new 60TB SSD is world’s largest, August 11, 2016. Retrieved February 2, 2018, from Ars Technica: <https://arstechnica.com/gadgets/2016/08/seagate-unveils-60tb-ssd-the-worlds-largest-hard-drive/>

2: AV-Test, Analysis of 160 Million Websites: Are Google and Other Search Engines Platforms for Distributing Malware?, October 26, 2016. Retrieved April 29, 2018, from AV-Test: <https://www.av-test.org/en/news/news-single-view/analysis-of-160-million-websites-are-google-and-other-search-engines-platforms-for-distributing-mal/>

3: Faroo FAQ, 2018. Retrieved February 2, 2018, from Faroo: <http://www.faroo.com/hp/p2p/faq.html>

4: Henzinger, M. Finding near-duplicate web pages: a large-scale evaluation of algorithms. *SIGIR '06 Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, 284-291. Retreived April 30, 2018 from epfl.ch: <https://infoscience.epfl.ch/record/99373/files/Henzinger06.pdf>

5: Introna, L. D. and Nissenbaum, H. Shaping the Web: Why

the Politics of Search Engines Matters. *The Information Society, 16,* (3), 169-185. DOI: <https://doi.org/10.1080/01972240050133634>

6: Noyes, K. YaCy: It's About Freedom, Not Beating Google, December 2, 2011. Retrieved April 29, 2018, from PC World: <https://www.pcworld.com/article/245414/yacy_its_about_freedom_not_beating_google.html>

7: Page, L. and Brin, S. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks and ISDN Systems,* *30* (1-7) 107-117. Retrieved from Stanford University: http://infolab.stanford.edu/pub/papers/google.pdf

8: Page, L., Brin, S., Motwani, R. and Winograd, T. 1998. *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report. Stanford University. Retrieved February 3, 2018, from CiteSeerx: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.38.5427&rep=rep1&type=pdf>

9: Safran, N. How the Filter Bubble Impacts Google, Bing Search & What It Means for SEOs, May 23, 2012. Retrieved February 2, 2018, from Search Engine Watch: <https://searchenginewatch.com/sew/how-to/2178874/filter-bubble-impacts-google-bing-search-means-seos>

10: Schwartz, B. Google’s search knows about over 130 trillion pages, November 14, 2016. Retrieved February 2, 2018, from Search Engine Land: <https://searchengineland.com/googles-search-indexes-hits-130-trillion-pages-documents-263378>

11: Wakabayashi, D. As Google Fights Fake News, Voices on the Margins Raise Alarm, September 26, 2017. Retrieved February 2, 2018, from The New York Times: <https://www.nytimes.com/2017/09/26/technology/google-search-bias-claims.html>

12: Weinberg, G. Google and Facebook are watching our every move online. It's time to make them stop, January 31, 2018. Retrieved February 2, 2018, from CNBC: <https://www.cnbc.com/2018/01/31/google-facebook-data-privacy-concerns-out-of-control-commentary.html>